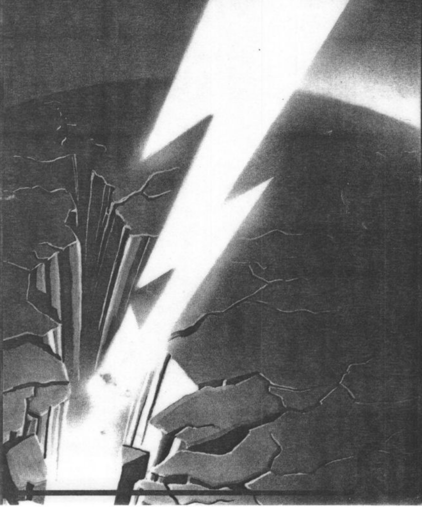
Programmeren van de

Z80

RODNAY ZAKS



programmeren van de Z80

programmeren van de Z80

C

3(

Za
Prc
Scl
Vei
ISB

Pro

programmeren van de Z80

RODNAY ZAKS

SYBEX

Originele uitgave in het engels
Titel van de engelse uitgave: Programming the Z80
Origineel copyright © SYBEK INC., Berkeley, Calif., USA
Druk: Drukkerij Salland BV, Deventer

Omslag ontwerp door **Daniel Le Noury**

Nederlandse vertaling door **Hans Scholten**

ISBN 3-88745-100-7
1. uitgave 1982

Printed in the Netherlands

DANKWOORD

Het schrijven van een boek dat over programmeren gaat is altijd moeilijk. Om het boek zo te maken dat het zowel het elementaire programmeren behandelt alsook geavanceerde programmeer technieken, terwijl het bovendien zowel hardware als software aan de orde laat komen, is een hele uitdaging. Voor hun suggesties voor verbeteringen en veranderingen dankt de schrijver: O. M. Barlow, Dennis L. Feick, Richard D. Reid, Stanley E. Erwin, Philip Hooper, Dennis B. Kitsz.
Speciaal is dank verschuldigd aan Chris Williams voor zijn bijdrage aan dat deel van het boek dat handelt over de instructieset en datastructuren.
Verdere suggesties voor verbeteringen en veranderingen kunnen naar de schrijver gezonden worden. Bij toekomstige drukken zal er rekening mee gehouden worden.

Enkele tabellen met codes voor de Z80- en 8080-instructies zijn overgenomen met toestemming van Zilog Inc, USA en Intel Corporation, USA.

(

3(

Za
Pr(
Sc
Ve
ISB

Pro

INHOUDSOPGAVE

VOORWOORD

Dit boek is een complete, op zichzelf staande cursus om te leren programmeren in assembler, waarbij gebruik wordt gemaakt van de Z80. Het is bedoeld zowel voor degene die nog nooit geprogrammeerd heeft, als voor degene die de Z80 al gebruikt.

Voor iedereen met programmeerervaring zal het boek ingaan op specifieke programmeertechnieken, gebruik makende van de eigenschappen van de Z80. Dit boek bestrijkt technieken van elementair tot gemiddeld niveau, welke nodig zijn, om te beginnen programmeren.

De bedoeling van dit boek is iedereen die een Z80 microprocessor wil programmeren te voorzien van de benodigde kennis en bekwaamheid. Natuurlijk is een boek alleen niet genoeg om te kunnen programmeren; daarvoor is oefening nodig. Men mag verwachten, dat dit boek de lezer zoveel zelfvertrouwen schenkt, dat deze zelf met programmeren begint,en zelfs vrij complexe problemen op kan lossen met behulp van een microcomputer.

Dit boek is gebaseerd op de ervaringen van de schrijver, opgedaan bij het leren programmeren aan meer dan 1000 personen. Het resultaat is een zeer gestructureerd boek. De hoofdstukken beginnen eenvoudig en worden geleidelijk moeilijker. Lezers met elementaire kennis over programmeren kunnen het inleidende hoofdstuk overslaan. Anderen, die nooit geprogrammeerd hebben, zullen de laatste paragrafen van bepaalde hoofdstukken meerdere keren moeten doornemen. Het boek laat de lezer systematisch kennis maken met alle basisbegrippen en technieken, nodig om steeds moeilijker programma's te maken. Daarom wordt ten zeerste aangeraden de volgorde van de hoofdstukken aan te houden. Bovendien is het belangrijk, om betere resultaten te bereiken, dat de lezer zoveel mogelijk oefeningen maakt. De moeilijkheidsgraad van de oefeningen is zorgvuldig opgebouwd. Met behulp van deze oefeningen kan men controleren of de tekst ook werkelijk begrepen is. Dit boek kan niet op zijn werkelijke waarde als leermiddel geschat worden,als de programmeeroefeningen niet worden gemaakt. Sommige oefeningen, zoals de vermenigvuldiging, kosten veel tijd, maar ze leren je wel programmeren.

Andere boeken uit deze serie behandelen het programmeren van andere populaire microprocessors.

Voor meer informatie over hardware raadplege men de boeken "From chips to systems: an introduction to microprocessors" en "Microprocessor interfacing techniques".

De inhoud van dit boek is zorgvuldig op fouten gecontroleerd, toch zullen ze onvermijdelijk voorkomen. De schrijver is zeer erkentelijk voor eventueel commentaar van lezers, zodat toekomstige drukken daar profijt van kunnen hebben. Iedere suggestie voor verbeteringen, of door de lezer geschreven belangrijke programma's worden zeer gewaardeerd.

1 BASIS BEGRIPPEN

INLEIDING

In dit hoofdstuk worden de basis begrippen en definities geintroduceerd die betrekking hebben op het programmeren met computers. De lezer die al bekend is met deze begrippen, zal misschien dit hoofdstuk even doorbladeren en dan door willen gaan naar hoofdstuk 2. Ik raad echter deze lezer aan ook dit hoofdstuk door te nemen. Veel wezenlijke begrippen komen aan de orde, zoals het "twee-complement","BCD", en andere representatie vormen. Sommige begrippen zullen nieuw zijn voor de lezer, en er komen misschien begrippen aan de orde waar zelfs de ervaren programmeur nog iets van kan leren.

WAT IS PROGRAMMEREN?

Voor een gegeven probleem moet eerst een oplossing gevonden worden. Deze oplossing in de vorm van een stap-voor-stap procedure wordt *algoritme* genoemd.Een algoritme is een stap-voor-stap specificatie van de oplossing voor een probleem. Het aantal stappen moet eindig zijn. Dit algoritme kan met iedere taal of symbolisme worden uitgedrukt. Een eenvoudig voorbeeld van een algoritme is:

1 - steek sleutel in sleutelgat
2 - draai sleutel een maal linksom
3 - pak de deurknop
4 - draai de knop linksom en duw tegen de deur

Op dat moment zal, als het algoritme juist is voor het gebruikte slot, de deur opengaan. Deze vier stappen grote procedure kan aangeduid worden als een algoritme om deuren te openen.

Wanneer de oplossing eenmaal is vastgelegd in een algoritme, moet het uitgevoerd worden door een computer. Helaas is het een vaststaand feit, dat computers normaal gesproken Nederlands niet begrijpen. De reden daarvoor is de *syntaktische dubbelzinnigheid* van alle gewone menselijke talen. Alleen een goed gedefinieerde deelverzameling van een natuurlijke taal kan door een computer "begrepen" worden. Deze deelverzameling heet dan een *programmeertaal.*

Programmeren is het omzetten van een algoritme in een reeks instructies van een programmeertaal. Om nauwkeuriger te zijn, de werkelijke vertaalfase van het algoritme naar de programmeertaal heet coderen. Programmeren heeft niet alleen betrekking op het coderen, maar ook op het ontwerpen van de programma's en de "datastructuren", die het programma uitvoeren.

Niet alleen kennis over mogelijke standaard algoritmes en computer hardware, maar ook een creatief gebruik van de juiste datastructuren is nodig om goed en effectief te programmeren. De volgende hoofdstukken zullen al deze aspecten behandelen.

Programmeren eist een grote discipline voor wat betreft de documentatie, zodat programma's niet alleen door de maker, maar ook door anderen te begrijpen zijn. Programma's moeten zowel intern als extern gedocumenteerd zijn.

Interne documentatie wil zeggen, dat in het programma commentaar staat, waaruit de werking van dat programma blijkt.

Externe documentatie heeft betrekking op alle documenten buiten het programma, zoals: geschreven verklaringen, handleidingen en stroomdiagrammen.

HET STROOMDIAGRAM

Als stap tussen *algoritme* en *programma* wordt vaak het *stroomdiagram* gebruikt. Een stroomdiagram is een eenvoudige symbolische representatie van het algoritme in de vorm van een opeenvolging van blokken, welke de verschillende stappen van het algoritme voorstellen. Daarbij worden rechthoeken gebruikt om commando's of "uit te voeren instructies" weer te geven. Ruiten worden gebruikt voor testen, zoals: als informatie A waar is, doe dan X, anders doe Y. Een formele definitie zal later, als we bezig zijn met programma's, aan de orde komen. Het maken van een stroomdiagram raad ik ten zeerste

aan als stap tussen algoritme en programma. Opmerkelijk is, dat misschien 10% van alle programmeurs programma's kunnen maken zonder stroomdiagram. Helaas is het ook zo, dat 90% denkt tot die 10% te horen. Gevolg: 80% van alle programma's bevatten gemakkelijk te voorkomen fouten, wanneer ze uitgevoerd worden op de computer. (Deze percentages moet u zien als een indicatie). Om kort te gaan, de meeste programmeurs zien het nut van een stroomdiagram niet in. Een fout programma is dan meestal het gevolg, en er moet veel tijd worden besteed om het programma te testen en te verbeteren (dit wordt "debuggen" of Foutzoeken genoemd). In elk geval is het aan te raden een stroomdiagram te maken.
Het kost weinig tijd meer dan het coderen alleen, en het resultaat is een programma dat snel foutloos is en doet wat het doen moet. Een klein aantal programmeurs zal, als ze het maken van een stroomdiagram onder de knie hebben, dit uit het hoofd kunnen doen, zonder iets op papier te zetten. Helaas zijn dergelijke programma's voor anderen vaak moeilijk te begrijpen zonder verdere documentatie. Maak daarom voor ieder belangrijk programma een stroomdiagram. Voorbeelden zijn het hele boek door te vinden.

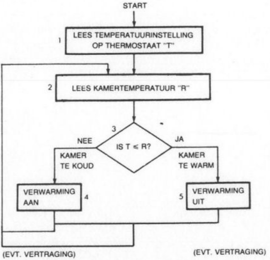


Fig. 1.1: Een stroomdiagram om de kamertemperatuur constant te houden

VOORSTELLING VAN INFORMATIE

Alle computers verwerken informatie in de vorm van cijfers of letters. We bekijken hier de interne en externe voorstelling van de informatie in een computer.

INTERNE VOORSTELLING VAN INFORMATIE

Alle informatie wordt in een computer opgeslagen in de vorm van groepen *bits*. Bit is afgeleid van het Engelse "binary digit", wat binair cijfer betekent (d.w.z. "0" of "1"). De twee toestanden die gewoonlijk in digitale circuits voorkomen zijn "aan" en "uit". Deze toestanden worden voorgesteld door de symbolen "1" en "0". Deze digitale circuits worden ook wel *binaire* circuits genoemd. Het gevolg is, dat tegenwoordig vrijwel alle informatie-verwerking gebeurt in het binaire formaat. Bij microprocessors in het algemeen, en bij de Z80 in het bijzonder, worden de bits in groepen van acht geplaatst. Zo'n groep van acht bits heet een *byte*. Een *nibble* is een groep van vier bits.

We gaan nu bekijken hoe de informatie intern wordt voorgesteld in dit binaire formaat. In de computer moeten twee dingen worden voorgesteld. Het eerste is het programma, dat wordt gevormd door een reeks instructies. Het tweede bestaat uit de data, waarop het programma bewerkingen uitvoert. De data bestaan uit getallen en alfanumerieke tekst. In de volgende onderdelen van dit hoofdstuk bespreken we de drie informatievormen: programma, getallen en alfanumerieke tekst.

Het programma

Alle instructies worden intern gevormd door een of meerdere bytes. Een zogeheten verkorte instructie is een byte lang. Een langere instructie wordt voorgesteld door twee of meer bytes. Omdat de Z80 een acht-bits microprocessor is, haalt deze een byte per keer uit het geheugen. Daarom kan een een-byte-instructie sneller worden uitgevoerd dan een twee- of drie-bytes-instructie. We zullen laten zien, dat dit een belangrijk punt is bij iedere microprocessor, en zeker bij de Z80, waar veel moeite is gedaan zo veel mogelijk verkorte instructies te maken, om de programma's efficienter te maken. Het beperken van de lengte tot acht bits heeft ook geleid tot belangrijke beperkingen, die later besproken zullen worden. Het is het klassieke voorbeeld van het compromis tussen snelheid en flexibiliteit. De codes voor de instructies zijn vastgelegd door de fabrikant. De Z80 is, net als iedere andere micro-

processor, voorzien van een vaste instructieset. Deze instructies zijn gedefinieerd door de fabrikant en zijn, tesamen met hun codes, te vinden achterin dit boek. Ieder programma bestaat uit een opeenvolging van deze binaire instructies. De Z80 instructies komen in hoofdstuk 4 aan de orde.

Numerieke data

De voorstelling van getallen is niet zo eenvoudig, en we moeten meerdere gevallen onderscheiden. Ten eerste moeten we gehele getallen voorstellen. Ten tweede getallen met een teken (d.w.z. positieve en negatieve getallen). En tenslotte decimale getallen (d.w.z. getallen met een komma). In het volgende komen deze eisen aan de orde met hun mogelijke oplossingen.

M.b.v. een directe binaire voorstelling kunnen we gehele getallen maken. Hiertoe wordt het getal rechtstreeks omgezet in een binair getal. In het binaire talstelsel is het meest rechtse bit 2 tot de macht 0. Het op een na rechtse bit 2 tot de macht 1, en het bit links daarvan 2 tot de macht 2. Het linkse bit is dan 2 tot de macht 7 = 128.

$$b_7b_6b_5b_4b_3b_2b_1b_0$$

representeert

$$b_72^7 + b_62^6 + b_52^5 + b_42^4 + b_32^3 + b_22^2 + b_12^1 + b_02^0$$

De machten van 2 zijn:
$2^7 = 128,\ 2^6 = 64,\ 2^5 = 32,\ 2^4 = 16,\ 2^3 = 8,\ 2^2 = 4,\ 2^1 = 2,\ 2^0 = 1$

De binaire voorstelling is analoog aan de normale decimale voorstelling, zoals:

$$\begin{array}{r} 1 \times 100 = 100 \\ + 2 \times 10 = 20 \\ + 3 \times 1 = 3 \\ \hline = 123 \end{array}$$

N.B. $100 = 10^2$, $10 = 10^1$, $1 = 10^0$.

In deze positieafhankelijke notatie komt ieder cijfer overeen met een macht van 10. In het binaire stelsel komt ieder cijfer of "bit" overeen met een macht van 2.

Bijvoorbeeld:
"00001001" binair weergegeven:

$$
\begin{array}{rl}
1 \times \quad 1 = 1 & (2^0) \\
0 \times \quad 2 = 0 & (2^1) \\
0 \times \quad 4 = 0 & (2^2) \\
1 \times \quad 8 = 8 & (2^3) \\
0 \times \ 16 = 0 & (2^4) \\
0 \times \ 32 = 0 & (2^5) \\
0 \times \ 64 = 0 & (2^6) \\
0 \times 128 = 0 & (2^7) \\
\hline
\end{array}
$$

= 9 in het decimale stelsel.

Nog een voorbeeld:
binair is: "10000001"

$$
\begin{array}{r}
1 \times \quad 1 = \quad 1 \\
0 \times \quad 2 = \quad 0 \\
0 \times \quad 4 = \quad 0 \\
0 \times \quad 8 = \quad 0 \\
0 \times \ 16 = \quad 0 \\
0 \times \ 32 = \quad 0 \\
0 \times \ 64 = \quad 0 \\
1 \times 128 = 128 \\
\hline
\end{array}
$$

= 129 decimaal.

Het is nu ook duidelijk, als we nog eens naar de binaire voorstelling kijken, waarom de bits van 0 tot 7 genummerd zijn. Bit 0 is b0 en hoort bij 2^0. Bit 1 is b1 en hoort bij 2^1, enzovoort.

De binaire equivalenten van de getallen 0 tot 255 staan in figuur 1.2.

Opgave 1.1: Wat is de decimale waarde van "11111100"?

Decimaal	Binair	Decimaal	Binair
0	00000000	32	00100000
1	00000001	33	00100001
2	00000010	•	
3	00000011	•	
4	00000100	•	
5	00000101	63	00111111
6	00000110	64	01000000
7	00000111	65	01000001
8	00001000	•	
9	00001001	•	
10	00001010	127	01111111
11	00001011	128	10000000
12	00001100	129	10000001
13	00001101		
14	00001110	•	
15	00001111	•	
16	00010000		
17	00010001	•	
•			
•			
•		254	11111110
31	00011111	255	11111111

Fig. 1.2: Decimaal-binair tabel

Van decimaal naar binair

We kunnen, omgekeerd, ook decimale getallen omzetten in binaire. Zoals bijvoorbeeld het decimale getal "11":

$11 \div 2 = 5$ rest $1 \longrightarrow 1$ (minst betekenisvolle bit)
$5 \div 2 = 2$ rest $1 \longrightarrow 1$
$2 \div 2 = 1$ rest $0 \longrightarrow 0$
$1 \div 2 = 0$ rest $1 \longrightarrow 1$ (meest betekenisvolle bit)

Het binaire equivalent van "11" decimaal is "1011". Dat is de rechtse kolom van beneden naar boven. Een decimaal getal kan in een binair getal omgezet worden, door het steeds door 2 te delen, totdat een quotient 0 verkregen is.

Opgave 1.2: Zet 257 om in een binair getal.

Opgave 1.3: Zet 19 om in een binair getal en weer terug in een decimaal.

Operaties op binaire data

De rekenregels voor binaire getallen zijn eenvoudig. De regels voor optellen zijn:

0+0=	0
0+1=	1
1+0=	~~0~~ 1
1+1=(1)	0

Een (1) betekent carry (of overdracht). Merk op, dat "10" binair hetzelfde is als "2" decimaal. Aftrekken gebeurt door het "complement" op te tellen. Dit zal verder verklaard worden als we weten hoe we negatieve getallen kunnen voorstellen.

Voorbeeld:

```
  (2)      10
 +(1)     +01
 ----     ---
 =(3)      11
```

Bij een optelling worden, net als in het decimale stelsel, de kolommen opgeteld, van rechts naar links.

Optelling van de rechtse kolom:

```
   10
  +01
  ---
```

(0 + 1 = 1. Geen carry)

Optelling van de volgende kolom:

```
 10
+01
---
 11     (1 + 0 =1. Geen carry)
```

Opgave 1.4: Bereken 5 + 10 binair en controleer dat de uitkomst 15 is.

Een paar andere voorbeelden:

```
 0010   (2)          0011   (3)
+0001   (1)         +0001   (1)
-----------         -----------
=0011   (3)         =0100   (4)
```

Het laatste voorbeeld laat de rol die de carry speelt zien.

Bekijk de rechtse bits: 1 + 1 = (1) 0. Er wordt een carry gegenereerd, die bij de volgende bits moet worden opgeteld.

```
   001 – kolom 0 is al opgeteld.
  +000 –
  +  1  (carry)
-----------
=  (1)0 – een nieuwe carry naar kolom 2
```

Het uiteindelijke resultaat is: 0100

Voorbeeld:

```
 0111      (7)
+0011    + (3)
--------------
 1010    =(10)
```

In dit voorbeeld vindt er een carry plaats naar de linkse kolom.

Opgave 1.5: Bereken:

```
 1111
+0001
-----
=?
```

Past het resultaat in 4 bits?

Op deze manier kunnen we met 8 bits alle getallen van "00000000" tot "11111111" voorstellen, d.w.z. van "0" tot "255". Twee hindernissen zijn direct zichtbaar. Ten eerste: we kunnen alleen positieve getallen weergeven. Ten tweede: bij gebruik van 8 bits is de grootte van het getal beperkt tot 255. Laten we deze hindernissen eens bekijken.

Binair met teken

Bij de binair met teken voorstelling wordt het meest linkse bit als tekenbit gebruikt. Traditioneel betekend een "0" een positief getal, en een "1" een negatief. Op deze manier is "11111111" "−127" decimaal, terwijl "01111111" "+127" decimaal is. We kunnen dus nu positieve en negatieve getallen voorstellen, maar de absolute grootte is nu beperkt tot 127.

Voorbeeld: "00000001" geeft +1 weer
"10000001" geeft −1 weer
Het tekenbit bij +1 is dus 0, bij −1 is dit 1.

Opgave 1.6: Wat is de binair met teken voorstelling van "−5"?

Nu het probleem van de grootte van het getal. Om grote getallen voor te stellen is het noodzakelijk veel bits te gebruiken. Als we bijvoorbeeld 16 bits nemen, kunnen we getallen van −32K tot +32K voorstellen (1K betekent in computer jargon 1024). Bit 15 bevat het teken en bits 14 tot 0 bevatten de absolute waarde. Wanneer dat nog te klein is kunnen we 3 of meer bytes nemen. Hoe groter de getallen, des te meer bytes zijn er nodig. Daarom hebben de meest eenvoudige versies van BASIC of PASCAL slechts een beperkt bereik voor gehele getallen. Betere versies gebruiken meer bytes, en kunnen dus grotere getallen weergeven.

Dan is er nog een ander probleem dat opgelost moet worden: de snelheid. We proberen eens twee getallen binair bij elkaar op te tellen. bijvoorbeeld "−5" en "+7".

+7 wordt weergegeven als	00000111
−5 wordt weergegeven als	10000101
De som binair weergegeven is:	10001100, of −12 decimaal

Het resultaat klopt niet. Dat moet +2 zijn, i.p.v. −12. Om toch het goede resultaat te krijgen met deze manier van voorstellen, moeten we enkele handelingen meer uitvoeren, die afhankelijk zijn van het teken. De optelling wordt complexer en dat komt de efficientie niet ten goede. Met andere woorden: een optelling binair met teken werkt niet zoals het zou moeten. Dat is vervelend. Om het nog eens duidelijk te stellen, de computer moet niet alleen de getallen voorstellen, maar er ook mee kunnen rekenen.

De oplossing voor dit probleem heet de twee-complement voorstelling, die gebruikt wordt i.p.v. de binair met teken voorstelling. Om het twee-complement duidelijker te maken, maken we eerst een tussenstap: het een-complement.

Een-complement

In het een-complement worden alle positieve getallen normaal binair voorgesteld. Dus "+3" wordt "00000011". Het complement, −3 dus, wordt verkregen door de binaire voorstelling van +3 te inverteren. D.w.z. iedere "0" wordt een "1" en omgekeerd. In een-complement wordt "−3": 11111100

Nog een voorbeeld:

+2 is 00000010
−2 is 11111101

Merk op dat positieve getallen beginnen met een "0" en negatieve met een "1".

Opgave 1.7: Decimaal "+6"is binair "00000110". Wat is "−6" in het een-complement?

Ter controle tellen we −4 en +6 op:

de som is:

−4 is 11111011
+6 is 00000110
(1) 00000001 (1) is een carry.

Het juiste antwoord moet "2" of "00000010" zijn. We proberen het nog een keer:

	−3 is	11111100
	−2 is	11111101
de som is	(1)	00000001

Dit resultaat is gelijk aan "1" plus een carry. Het had "−5" moeten zijn, binair met teken: "11111010". Ook hier klopt het niet.

Met deze voorstelling kunnen we positieve en negatieve getallen maken. Het resultaat van een optelling klopt echter niet. We zullen daarom nog een andere voorstelling gebruiken, welke ontwikkeld is uit het een-complement: het twee-complement.

Twee-complement voorstelling

M.b.v. het twee-complement worden positieve getallen op dezelfde manier voorgesteld als met de binair met teken voorstelling, die gelijk is aan de een-complement notatie. Het verschil zit in de negatieve getallen. Bij twee-complement krijgen we een negatief getal door eerst het een-complement te berekenen en er dan 1 bij op te tellen. Ter illustratie het volgende voorbeeld:

+3 binair met teken is 00000011
hiervan het een-complement: 11111100
tel hier 1 bij op: 11111101

We proberen eerst een optelling:

(3)	00000011
+(5)	+00000101
=(8)	=00001000

Het resultaat klopt.
Nu trekken we af:

(3)	00000011
(−5)	+11111011
	=11111110

Van dit antwoord berekenen we het twee-complement:

het een-complement is:	11111110 is	00000001
	tel er 1 bij op	+ 1
		00000010 of +2

Het antwoord is dus −2, wat klopt.

We hebben nu een optelling en een aftrekking gedaan en de antwoorden kloppen (afgezien van een eventuele carry). Het lijkt erop, dat het twee-complement werkt!

Opgave 1.8: Bereken het twee-complement van +127

Opgave 1.9: Wat is het twee-complement van −128?

Tel +4 en −3 op. (Doe dit door de twee-complementen op te tellen).

+4 is	00000100
−3 is	11111101
(1)	00000001

Het resultaat is 1, als we de carry negeren. Dat klopt dus. Zonder het wiskundige bewijs te leveren, volstaan we hier met de constatering dat deze wijze van voorstellen werkt. In het twee-complement is het mogelijk getallen met een teken op te tellen en af te trekken. Daarbij kan gebruik worden gemaakt van dezelfde rekenregels als van de normale binaire optelling. Het resultaat klopt altijd, inclusief het teken. Dit laatste is zeer belangrijk. Als dat namelijk niet het geval zou zijn geweest, dan zouden we correctiemethodes moeten toepassen om toch het juiste teken te krijgen. Het zou dus meer tijd kosten een optelling of een aftrekking uit te voeren.

Om volledig te zijn, moeten we toegeven dat de twee-complement voorstelling de meest gemakkelijke is voor eenvoudige computers, zoals microprocessors. Op meer complexe computers kunnen nog andere voorstellingen worden gebruikt. Dit kan het een-complement zijn, maar dan zijn speciale circuits nodig om het resultaat te corrigeren.

Vanaf nu zullen alle gehele getallen in dit boek worden genoteerd in het twee-complement. In figuur 1.3 staat een tabel voor twee-complement getallen.

Oefening 1.10: Wat is het grootste en het kleinste getal dat we nog kunnen voorstellen in twee-complement, gebruik makende van 1 byte?

Oefening 1.11: Bereken het twee-complement van 20. Bereken dan het twee-complement van het antwoord. Is dat weer 20?

De volgende voorbeelden dienen ter illustratie van de regels van het twee-complement. C betekent een mogelijke carry of borrow. C is bit 8, dus het negende bit, van het resultaat. V betekent een overflow, d.w.z. V wordt 1, als het teken per ongeluk verandert omdat het resultaat te groot is. In feite is het een interne carry van bit 6 naar bit 7, het tekenbit. Dit zal verduidelijkt worden in het nu volgende.

De carry C

Een voorbeeld van een carry:

```
   (128)          10000000
  +(129)         +10000001
  ------------------------
   (257) = (1)    00000001
```

(1) is een carry.

Het resultaat heeft een negende bit nodig (dit zou bit 8 worden). Dit negende bit wordt het carrybit genoemd. Het resultaat is dus 100000001 = 257. De carry moet echter als zodanig herkenbaar zijn en we moeten er dus zeer zorgvuldig mee omspringen. De registers in de microprocessor zijn in het algemeen slechts 8 bits breed. Van het resultaat wordt alleen bit 0 tot en met bit 7 bewaard. De carry moet met een speciale instructie gedetecteerd, en op de juiste wijze verder behandeld worden. Dat kan bijvoorbeeld zijn: opbergen van de carry, de carry negeren, of besluiten dat er een fout is opgetreden (Dit laatste als het grootste toegestane getal 11111111 is).

+	twee complement code	−	twee complement code
+127	01111111	−128	10000000
+126	01111110	−127	10000001
+125	01111101	−126	10000010
...		−125	10000011
		...	
+65	01000001	−65	10111111
+64	01000000	−64	11000000
+63	00111111	−63	11000001
...		...	
+33	00100001	−33	11011111
+32	00100000	−32	11100000
+31	00011111	−31	11100001
...		...	
+17	00010001	−17	11101111
+16	00010000	−16	11110000
+15	00001111	−15	11110001
+14	00001110	−14	11110010
+13	00001101	−13	11110011
+12	00001100	−12	11110100
+11	00001011	−11	11110101
+10	00001010	−10	11110110
+9	00001001	−9	11110111
+8	00001000	−8	11111000
+7	00000111	−7	11111001
+6	00000110	−6	11111010
+5	00000101	−5	11111011
+4	00000100	−4	11111100
+3	00000011	−3	11111101
+2	00000010	−2	11111110
+1	00000001	−1	11111111
+0	00000000		

Fig. 1.3: Twee-complement tabel

De overflow V

Een overflow treedt op in het volgende voorbeeld:

```
bit 6 ───────┐
bit 7 ──────┐│
            ▼▼
            01000000      (64)
           +01000001     +(65)
     ─────────────────────────
          =10000001   =(-127)
```

Een carry van bit 6 naar bit 7 is gegenereerd. Dit heet een overflow. Het resultaat is "per ongeluk" negatief geworden. Dit moet worden gedetecteerd om het resultaat te kunnen corrigeren.

We bekijken nu de volgende situatie:

```
          11111111     (-1)
         +11111111    +(-1)
     ─────────────────────────
=(1)      11111110    =(-2)
  ▼
carry
```

Er komt tweemaal een carry voor: een van bit 6 naar bit 7 en een van bit 7 naar het carrybit. De rekenregels voor het twee-complement vertellen ons, dat we de carry van bit 7 naar het carrybit kunnen negeren. Het resultaat klop zo. De interne carry immers veranderde het teken niet. Er is dus geen sprake van een overflow. Bij het werken met negatieve getallen is een interne carry dus niet eenvoudig een overflow.
Nog een voorbeeld:

```
          11000000     (-64)
         +10111111     (-65)
     ─────────────────────────
=(1)      01111111    (+127)
  ▼
carry
```

Er is geen interne, maar wel een externe carry. Het resultaat is fout, omdat bit 7 veranderd is. Hier is wél sprake van een overflow.

Een overflow kan voorkomen in de volgende situaties:

1 - bij het optellen van grote positieve getallen,
2 - bij het optellen van grote negatieve getallen,
3 - bij het aftrekken van een groot positief van een groot negatief getal,
4 - bij het aftrekken van een groot negatief van een groot positief getal.

We kunnen dus nu een betere definitie geven van een overflow:

Technisch bekeken is een overflow indicator een speciaal voor dit doel gereserveerd bit, een "vlag" genoemd, dat 1 gemaakt wordt, als er een carry plaats vindt van bit 6 naar bit 7, en er geen externe carry is, of als er een externe carry is en geen interne. Deze vlag geeft aan, dat het tekenbit, bit 7, per ongeluk veranderd is. Voor de technisch geinteresseerde lezer: de overflow vlag is de uitgang van een exclusieve of van de carry-in en de carry-uit van bit 7. Praktisch iedere microprocessor is met een dergelijke vlag uitgerust.

Een overflow geeft aan, dat het resultaat van een optelling of een aftrekking meer bits nodig heeft dan de 8 van een register.

De carry en de overflow

De carry en de overflow bits worden *vlaggen* genoemd. Ze komen voor in iedere microprocessor en in het volgende hoofdstuk zullen we ze leren gebruiken om effectief te kunnen programmeren. Deze twee indicatorbits zitten in een speciaal register, het vlag of "status" register. Dit register bevat nog meer indicatorbits, maar deze komen pas in hoofdstuk 4 aan de orde.

Voorbeelden

In de volgende voorbeelden wordt het gebruik van de carry en de overflow nog eens geillustreerd. V is overflow en C is carry. Is V = 0, dan is er geen overflow. Als V = 1 wel. Hetzelfde geldt voor C. Onthoudt goed, dat de rekenregels van het twee-complement aangeven, dat de carry genegeerd moet worden. (Het wiskundige bewijs daarvoor wordt hier niet geleverd).

Positief-positief

```
    00000110   (+6)
+   00001000   (+8)
=   00001110   (+14)   V:0     C:0
```

Correct

Positief-positief met overflow

```
    01111111   (+127)
+   00000001     (+1)
=   10000000   (-128)  V:1     C:0
```

Het resultaat is niet geldig, want er is een overflow.

Fout

Positief-negatief (positief resultaat)

```
    00000100   (+4)
+   11111110   (-2)
=(1)00000010   (+2)    V:0     C:1 (negeren)
```

Correct

Positief-negatief (negatief resultaat)

```
    00000010   (+2)
+   11111100   (-4)
=   11111110   (-2)    V:0     C:0
```

Correct

Negatief-negatief

```
    11111110   (-2)
+   11111010   (-4)
=(1)11111010   (-6)    V:0     C:1 (negeren)
```

Correct

Negatief-negatief met overflow

```
    10000001   (-127)
+   11000010    (-62)
=(1)01000011     (67)  V:1     C:1
```

Fout

Er treedt een overflow op, door twee grote negatieve getallen op te tellen. Het resultaat moet −189 zijn, wat te groot is voor 8 bits.

Oefening 1.12: Maak de volgende optellingen af. Noteer het resultaat, carry C, overflow V en geef aan of het resultaat klopt.

10111111 (____) +11000001 (____) = ________ V:____ C:____ ☐ CORRECT ☐ fout	11111010 (____) +11111001 (____) = ________ V:____ C:____ ☐ CORRECT ☐ fout
00010000 (____) +01000000 (____) = ________ V:____ C:____ ☐ CORRECT ☐ fout	01111110 (____) +00101010 (____) = ________ V:____ C:____ ☐ CORRECT ☐ fout

Oefening 1.13: Kun je een voorbeeld geven van een optelling van een positief en een negatief getal, waarbij een overflow optreedt? Waarom?

Vast formaat voorstelling

We weten nu hoe we gehele getallen met een teken kunnen voorstellen. Het grootte probleem is daarmee nog niet opgelost. Als grote getallen moeten worden voorgesteld, zijn meerdere bytes nodig. Het is echter efficienter om rekenkundige operaties uit te voeren met een vast aantal bytes, i.p.v. met een aantal dat kan varieren. Wanneer het aantal bytes is gekozen, is daarmee de maximale grootte van de getallen vastgelegd.

Oefening 1.14: Wat zijn de grootste en de kleinste getallen die te maken zijn met twee bytes in twee-complement voorstelling?

Het probleem van de grootte der getallen

We hebben ons bij het optellen beperkt tot getallen van acht bits, omdat dat het formaat is, waarop de processor intern bewerkingen uitvoerd. Daarmee zijn we ook beperkt tot getallen van −128 tot +127. Het is duidelijk dat dit voor veel toepassingen niet voldoende is.

Daarom wordt dubbele of zelfs meervoudige precisie gebruikt om meer cijfers voor te stellen. Twee, drie of N bytes zijn daarvoor te ge-

bruiken. Bijvoorbeeld het 16 bits "dubbele precisie" formaat:

00000000	00000000	is "0"
00000000	00000001	is "1"
...		
01111111	11111111	is "32767"
11111111	11111111	is "−1"
11111111	11111110	is "−2"

Oefening 1.15: Wat is het grootste negatieve getal in twee-complement in het drie-voudige precisie formaat?

Er zijn nadelen verbonden aan deze methode. In het algemeen kunnen bij een optelling slechts acht bits per keer worden opgeteld. Dit wordt verder verklaard in hoofdstuk 3 (Basis programmeer technieken). Een volledige optelling is nu langzamer geworden. Een ander nadeel is dat ook kleine getallen 16 bits gebruiken i.p.v. 8. Daarom worden zelden meer dan 16 of 32 bits gebruikt.

Dan is er nog het volgende belangrijke punt: welk aantal bits we ook nemen (stel dit aantal N), dit aantal kan daarna niet meer veranderd worden. Als het resultaat van een berekening meer dan N bits nodig heeft, zullen bits verloren gaan. Normaal bewaart het programma de meest significante bits, en de minst significante worden weggegooid. Dit heet afkappen.

Ter illustratie een voorbeeld in het decimale stelsel, waarin we 6 cijfers gebruiken:

$$\begin{array}{r} 123456 \\ \times \quad 1.2 \\ \hline 246912 \\ 123456 \\ \hline =148147.2 \end{array}$$

Het resultaat heeft 7 cijferplaatsen nodig! De 2 achter de komma wordt weggegooid en het uiteindelijke resultaat is 148147. Het wordt dus afgekapt. Zolang de positie van de komma niet verloren gaat, is dit een gebruikelijke manier om het bereik van de operaties te vergroten. Dit gaat wel ten koste van de precisie.

In het binaire stelsel geldt het zelfde probleem. De vermenigvuldiging komt in hoofdstuk 4 aan de orde.

De vaste formaat voorstelling kan een verlies van de precisie tot gevolg hebben, maar voor gewone en wiskundige berekeningen kan deze voorstelling voldoen.

Helaas is het verlies van precisie niet toegestaan bij boekhoudkundige operaties. Stel je voor, dat ieder bedrag van 5 cijfers afgerond zou worden op hele guldens. Niemand accepteert dat.

Waar precisie essentieel is, moeten we iets anders verzinnen. De meest gebruikte oplossing is BCD (afkorting van het engelse Binary Coded Decimal = binair gecodeerde decimale voorstelling).

BCD (Binary Coded Decimal)

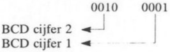
Bij BCD wordt ieder decimaal cijfer afzonderlijk binair gecodeerd, waarbij zoveel bits worden gebruikt als nodig zijn om het getal exact weer te geven. Voor ieder cijfer zijn 4 bits nodig. Drie bits zijn niet voldoende, want dan zouden we alleen de getallen 0 tot en met 7 kunnen weergeven. Met 4 bits zijn 16 combinaties te maken, wat meer dan voldoende is voor de getallen 0 tot en met 9. Zes combinaties worden niet gebruikt. Zie figuur 1.4. Dit geeft problemen bij optellingen en aftrekkingen.

CODE	BCD SYMBOOL	CODE	BCD SYMBOOL
0000	0	1000	8
0001	1	1001	9
0010	2	1010	ongebruikt
0011	3	1011	ongebruikt
0100	4	1100	ongebruikt
0101	5	1101	ongebruikt
0110	6	1110	ongebruikt
0111	7	1111	ongebruikt

Fig. 1.4: BCD tabel

Omdat we per cijfer 4 bits nodig hebben, passen twee BCD cijfers in een byte. Dat heet "packed BCD".
Voorbeeld:

Een BCD code zit als volgt in elkaar:



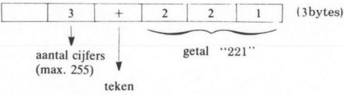
Het BCD getal is dus 21

Oefening 1.16: Wat zijn de BCD voorstellingen van 29 en 91?

Oefening 1.17: Is 10100000 een geldige BCD voorstelling? Waarom?

Zoveel bytes als nodig zijn worden gebruikt om alle BCD cijfers voor te stellen. Een of twee nibbles aan het begin geven het totale aantal nibbles aan van het getal, d.w.z. het aantal cijfers van de BCD voorstelling. Een andere nibble geeft de plaats van de komma aan. Afspraken daarover kunnen echter varieren.

Een voorbeeld van een BCD getal:



(De code voor het teken is b.v. 0000: +; 0001: −)

Oefening 1.18: Wat is het getal "−23123" in het BCD formaat volgens de afspraken van bovenstaand voorbeeld? En wat is de binaire voorstelling?

Oefening 1.19: Zet "222" en "111" om in BCD code. Doe hetzelfde voor het resultaat van "222 × 111".

De BCD voorstelling kan gemakkelijk getallen met een komma weergeven, zoals bijvoorbeeld +2,21:

3	2	+	2	2	1

3 cijfers — komma links van het tweede cijfer — + — 221

Het voordeel van BCD is, dat het resultaat precies is. Een nadeel is de grote hoeveelheid geheugen die nodig is en de trage rekenkundige verwerking. Dit is alleen acceptabel waar deze nauwkeurigheid vereist is, zoals in boekhoudingen. In andere gevallen wordt BCD niet gebruikt.

Oefening 1.20: Hoeveel bits zijn nodig om "9999" in BCD te coderen? En hoeveel in het twee-complement?

We hebben nu de problemen die samenhangen met het weergeven van gehele getallen, getallen met een teken, en zelfs grote getallen opgelost. We hebben ook al een manier leren kennen om decimale getallen voor te stellen, n.l. BCD. De volgende stap is de voorstelling van decimale getallen in een vast formaat.

Floating-point voorstelling

Het basis principe is, dat ook decimale getallen m.b.v. een vast aantal bytes weergegeven moeten worden. Om geen bits te verspillen worden de getallen genormaliseerd.

Zo verspilt b.v. "0,000123" vier nullen links in het getal, die alleen maar nodig zijn om aan te geven waar de komma staat. Hetzelfde getal, maar nu genormaliseerd is ",123 $\times$ 10^{-3}". ",123" heet de *genormaliseerde mantisse*, "−3" is de *exponent*. Het getal is dus genormaliseerd door alle overbodige nullen links in het getal weg te laten en de exponent aan te passen.

Een ander voorbeeld:

22,1 wordt ,221 $\times$ 10^2

Of in het algemeen: $M \times 10^E$; M = mantisse, E = exponent.

Het is gemakkelijk in te zien, dat een genormaliseerd getal gekenmerkt wordt door een mantisse kleiner dan 1, maar groter dan of gelijk aan 0,1. Dus

$$.1 \leqslant M < 1 \text{ of } 10^{-1} \leqslant M < 10^{0}$$

of

$$2^{-1} \leqslant M < 2^{0} \text{ (of } .5 \leqslant M < 1)$$

Voor M is de absolute waarde genomen van de mantisse (we kijken dus niet naar het teken).

Voorbeeld:

$$111{,}01 \text{ wordt } ,11101 \times 2^{3}$$

M = 11101, E = 3

We kennen nu het principe, maar hoe ziet zo'n floating-point voorstelling er in het echt uit? Figuur 1.5 geeft een mogelijk formaat.

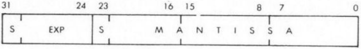


Fig. 1.5: Floating-point voorstelling

Zoals de figuur laat zien, worden vier bytes of 32 bits gebruikt. De linkse byte bevat de exponent met zijn teken. Zowel de exponent als de mantisse zijn in het twee-complement. De maximale exponent is "−128". De mantisse met teken neemt drie bytes in beslag: een bit voor het teken, 23 bits voor de grootte.

Oefening 1.21: Hoeveel decimale cijfers kun je met de 23 bits van de mantissa weergeven?

Dit is slechts een voorbeeld van een floating-point voorstelling. Het is natuurlijk mogelijk meer of minder bytes te gebruiken dan de vier in het voorbeeld. Vier bytes echter vormen een goed compromis tussen nauwkeurigheid, grootte van de getallen, gebruik van geheugen en de efficientie van rekenkundige operaties.

We hebben gezien hoe we getallen kunnen voorstellen. Laten we nu

een kijkje nemen in de microprocessor, om te zien hoe daar alfanumerieke data wordt voorgesteld.

Voorstelling van alfanumerieke data

De voorstelling van alfanumerieke data, d.w.z. karakters, is recht-toe-recht-aan: alle karakters worden gecodeerd naar acht bits. In de computer wereld zijn twee codes in gebruik: De ASCII en de EBCDIC code. ASCII (afkorting van "American Standard Code for Information Interchange") is algemeen gangbaar bij microprocessors. EBCDIC is een variatie op ASCII en wordt door IBM gebruikt. De enige keer dat we iets met EBCDIC te maken zullen krijgen is wanneer we een IBM terminal aan willen sluiten.

In het kort gaan we ASCII bekijken. Er moeten 26 letters, hoofdletters en kleine letters, plus 10 numerieke symbolen, en misschien nog eens 20 speciale symbolen gecodeerd worden. Dit kan gemakkelijk met 7 bits gedaan worden, want daarmee zijn 128 combinaties te maken. Het achtste bit is, als het gebruikt wordt, het *pariteitsbit*. M.b.v. de pariteit is te controleren of de inhoud van een byte niet per ongeluk veranderd is. Het aantal enen in een byte wordt geteld en is dit aantal oneven, dan wordt het achtste bit een 1. Het aantal enen wordt dus even gemaakt. Dit heet een even pariteit. Op dezelfde manier is een oneven pariteit te maken: het pariteitsbit wordt zodanig gemaakt, dat het aantal enen oneven is.

Voorbeeld: Bereken het pariteitsbit van "0010011" bij gebruik van een even pariteit. Er zijn 3 enen. Het pariteitsbit wordt 1, waarmee het totale aantal enen 4 is geworden, dus even. Het resultaat is "10010011", met het linkse bit als pariteit, terwijl "0010011" de code voor het karakter is.

De tabel met de 7 bits ASCII codes staat in figuur 1.6. In de praktijk wordt de code vaak zonder pariteit gebruikt. Het achtste bit is dan altijd 0. Is er wel een pariteitsbit, dan is dit altijd het linkse bit van een byte

Oefening 1.22: Bereken de 8 bits code voor de getallen 0 tot en met 9 met even pariteit. (Deze codes worden gebruikt in enkele voorbeelden van toepassingen in hoofdstuk 8).

Oefening 1.23: Doe hetzelfde voor de letters A tot en met F.

Oefening 1.24: Geef aan wat de binaire inhoud is van de bytes "A" "?" "3" "6", gebruik makende van ASCII code zonder pariteit (het pariteitsbit is 0).

HEX	MSD	0	1	2	3	4	5	6	7
LSD	BITS	000	001	010	011	100	101	110	111
0	0000	NUL	DLE	SPACE	0	@	P	–	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	--
D	1101	CR	GS	–	=	M	]	m	}
E	1110	SO	RS	.	>	N	∧	n	~
F	1111	SI	US	/	?	O	←	o	DEL

Fig. 1.6: ASCII tabel. (zie bijlage B voor de afkortingen)

In speciale gevallen, telecommunicatie bijvoorbeeld, kunnen andere codes voorkomen, zoals fout-verbeterende codes. Deze liggen echter buiten het bestek van dit boek.

We hebben de gebruikelijke voorstellingsvormen in de computer van zowel programma als data bekeken. Nu komen de externe voorstellingen aan de orde.

EXTERNE VOORSTELLING VAN INFORMATIE

De externe voorstelling heeft betrekking op de wijze waarop informatie, meestal aan de programmeur, gepresenteerd wordt. Dit gebeurt in hoofdzaak in drie mogelijke formaten: binair, octaal of hexadecimaal en symbolisch.

Binair

Het is al aan de orde gekomen, dat informatie in de computer wordt opgeslagen in de vorm van bytes. Soms is het wenselijk de informatie in deze vorm te laten zien. Dit is de *binaire voorstelling*. LED's (Light Emitting Diodes= licht uitstralende diodes) op het bedieningspaneel van een computer zijn hiervan een voorbeeld. Bij een 8 bits microprocessor zullen dat 8 LED's zijn, waarmee bijvoorbeeld de inhoud van registers zichtbaar is te maken. (Registers worden behandeld in hoofdstuk 2). Een brandende LED geeft een 1 aan, een niet brandende LED een 0. De binaire voorstelling kan gebruikt worden voor het foutzoeken in complexe programma's, vooral als er sprake is van I/O (I/O = input/output = invoer / uitvoer), maar het is natuurlijk onpraktisch op menselijk niveau. D.w.z. wij kijken liever naar informatie in een symbolische vorm: "9" is gemakkelijker te begrijpen en te onthouden dan "1001". Eenvoudiger voorstellingen zijn daarom ontworpen om de communicatie tussen mens en machine gemakkelijker te maken.

Octaal en hexadecimaal

"Octaal" en "hexadecimaal" coderen drie resp. vier bits in een uniek symbool. In het octale systeem wordt iedere combinatie van drie bits voorgesteld door een getal tussen θ en 7. Zie figuur 1.7: Octale symbolen.

binair	octaal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Fig.1.7: Octale symbolen

Voorbeeld:

"00 100 100" binair is
▼ ▼ ▼
" 0 4 4 " octaal

"11 111 111" binair is
▼ ▼ ▼
" 3 7 7 " octaal

Omgekeerd: " 2 1 1 " octaal is
"10 001 001" binair

Het octale stelsel wordt traditioneel gebruikt in de oudere computers die werkten met een variabel aantal bits. Dit aantal kon liggen tussen 8 en 64. Tegenwoordig, nu de 8 bits microprocessors de overhand krijgen, is het 8 bits formaat standaard geworden en wordt een praktischer voorstelling gebruikt: het hexadecimale systeem.

In dit systeem wordt een groep van vier bits gecodeerd in een hexadecimaal cijfer. Deze cijfers zijn de getallen 0 tot en met 9, en de letters A, B, C, D, E en F. Zo wordt bijvoorbeeld "0000" binair "0" hexadecimaal, "0001" wordt "1" en "1111" wordt "F". (Zie figuur 1.8).

DECIMAAL	BINAIR	HEX	OCTAAL
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	8	10
9	1001	9	11
10	1010	A	12
11	1011	B	13
12	1100	C	14
13	1101	D	15
14	1110	E	16
15	1111	F	17

Fig. 1.8: Hexadecimale codes

Voorbeeld:

1010 0001 binair is
A 1 hexadecimaal

Oefening 1.25: Wat is de hexadecimale representatie van "10101010"?

Oefening 1.26: Wat is het binaire equivalent van "FA"?

Oefening 1.27: Wat is de octale code voor "01000001" binair?

Hexadecimale voorstelling heeft het voordeel dat 8 bits weergegeven worden m.b.v. 2 symbolen. Het is gemakkelijker te herkennen en te onthouden, en bovendien is het sneller in te typen. Daarom is bij de meeste nieuwe microcomputers de hexadecimale voorstelling de meest gebruikte om de inhoud van registers of het geheugen weer te geven.

Wanneer de inhoud van een register of een geheugen plaats een speciale betekenis heeft, als het b.v. letters bevat, zal de hexadecimale voorstelling niet zo gemakkelijk voor de mens te begrijpen zijn.

Symbolische voorstelling

Symbolische voorstelling heeft betrekking op het weergeven van informatie in symbolische vorm. Zo zullen, bijvoorbeeld, decimale getallen als decimale getallen voorgesteld worden en niet als een reeks hexadecimale cijfers of als bits. Op dezelfde wijze wordt tekst als tekst weergegeven. Symbolische voorstelling is natuurlijk de meest praktische voor de gebruiker. Wanneer een geschikt medium, zoals een beeldscherm of een printer, beschikbaar is, zal deze vorm van voorstelling gebruikt worden. Helaas is het voor kleinere systemen economisch niet verantwoord deze te voorzien van beeldschermen of printers, waardoor de gebruiker gebonden is aan een communicatie met de processor in hexadecimale vorm.

Samenvatting externe voorstellingen

De meest wenselijke vorm van voorstelling van informatie is de symbolische. Omdat daarvoor een toetsenbord en een beeldscherm of een printer nodig zijn is dat echter nogal duur. Daarom zal deze vorm van voorstellen niet voorkomen op de minder dure systemen. Een alterna-

tief moet dan worden gebruikt. In de meeste gevallen is dat de hexadecimale voorstelling. Slechts in uitzonderlijke gevallen, bijvoorbeeld tijdens het foutzoeken, gebeurt de voorstelling binair. In dat geval wordt de inhoud van een register of het geheugen in binaire vorm weergeven.

(Het gebruik van binaire displays op het bedieningspaneel van computers is altijd al het onderwerp geweest van heftige en emotionele debatten. In dit boek zullen we daar niet verder op in gaan).

We hebben nu bekeken hoe interne en externe informatie voorgesteld wordt. In het volgende hoofdstuk komt de eigenlijke microprocessor, die met de informatie moet werken, aan de orde.

Aanvullende oefeningen

Oefening 1.28: Wat is het voordeel van het twee-complement boven andere vormen van voorstellen bij het weergeven van getallen met een teken?

Oefening 1.29: Hoe zou je "1024" binair voorstellen? En binair met teken? En hoe in het twee-complement?

Oefening 1.30: Wat is het V-bit? Moet de programmeur dit bit voor of na een optelling of aftrekking controleren?

Oefening 1.31: Bereken het twee-complement van: +16, +17, +18, −16, −17 en −18.

Oefening 1.32: Geef de hexadecimale voorstelling van de volgende tekst, die intern in het ASCII formaat zonder pariteitsbit is opgeborgen: = "BOODSCHAP".

2
Z80 HARDWARE ORGANISATIE

INLEIDING

Om op een elementair niveau te kunnen te programmeren is het niet nodig de interne structuur van de processor tot in detail te kennen. Om efficient te programmeren is deze kennis zeker nodig. In dit hoofdstuk worden de hardware beginselen gepresenteerd, die nodig zijn om de werking van een Z80 microprocessor systeem te begrijpen. Een compleet systeem bevat niet alleen de microprocessor eenheid, hier de Z80, maar ook andere componenten. In dit hoofdstuk komt alleen de Z80 aan de orde, terwijl in hoofdstuk 7 de andere apparatuur (vooral I/O = Input/Output = invoer/uitvoer) besproken wordt.

We zullen eerst de basisarchitectuur van het microprocessorsysteem bekijken, waarna we de interne organisatie van de Z80 onder de loep nemen, in het bijzonder de verschillende registers. Daarna komt de wijze waarop een programma wordt uitgevoerd aan de orde, tesamen met het mechanisme dat zorgt voor de juiste opeenvolging van de diverse signalen. Vanuit een hardware standpunt uit bekeken is dit hoofdstuk echter een vereenvoudigde weergave. De lezer die de hardware meer gedetailleerd wil leren kennen verwijs ik naar het boek "Microprocessors van chip tot systeem".

De Z80 was ontworpen als vervanging van de Intel 8080, met enkele mogelijkheden meer. In dit hoofdstuk wordt een aantal verwijzingen naar het 8080 ontwerp gemaakt.

SYSTEEM ARCHITECTUUR

De architectuur van het microprocessorsysteem staat in figuur 2.1. De microprocessor eenheid (MPU = Microprocessor Unit) is links in de figuur te vinden. Het verenigt alle functies van een centrale verwerkingseenheid (CPU = Central Processing Unit) in een chip: een rekenkundige-logische eenheid (ALU = Arithmetic-Logical Unit), plus de interne registers en een controle eenheid (CU = Control Unit), die zorgt voor de juiste volgorde van de verschillende handelingen. De werking wordt in dit hoofdstuk verklaard.

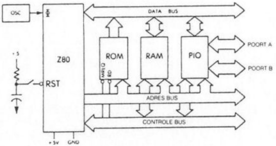


Fig. 2.1: Standaard Z80 systeem

De MPU heeft drie *bussen*: een 8 bits tweerichtings *databus*, boven in de figuur; een 16 bits eenrichtings *adresbus* en een *controlebus*, beide laatsten onder in de figuur. De functie van ieder van deze bussen wordt nu beschreven.

Over de *databus* worden de data uitgewisseld tussen de verschillende elementen van het systeem. Hoofdzakelijk tussen MPU en geheugen, en tussen MPU en I/O chips (een I/O chip is een component die zorgt voor de communicatie met de externe apparatuur).

De *adresbus* transporteert een door de MPU gegenereerd adres, waarmee een intern register in een van de chips in het systeem geselecteerd wordt. Dit adres bepaalt de bron of de bestemming van de over de databus getransporteerde data.

De *controlebus* bevat de synchronisatie signalen die nodig zijn in het systeem.

Nu het doel van de bussen beschreven is, kunnen we de componenten, nodig om een compleet systeem te maken, met elkaar gaan verbinden.

Iedere MPU heeft een preciese timing nodig. Deze wordt verzorgd door een klok en een kristal. In de meeste "oudere" microprocessors is de klokoscillator extern van de MPU en is er een aparte chip voor nodig. In de nieuwere processors zit de oscillator in de MPU. Het kristal echter is, vanwege de grootte, nog altijd extern. De oscillator staat links van de MPU in figuur 2.1.

We zullen nu de andere elementen van het systeem de revue laten passeren. Van links naar rechts zijn in de illustratie te onderscheiden:

De ROM: is een geheugen waarin alleen gelezen kan worden (ROM = Read Only Memory). Het bevat het programma van het systeem. Het voordeel van ROM is, dat de inhoud permanent is en niet verdwijnt als het systeem wordt uitgezet. De ROM bevat daarom altijd een *bootstrap* of *monitor* (dit zal later worden verklaard), waarmee het systeem opgestart kan worden als het wordt aangezet. In toepassingen waar het programma nooit wordt veranderd, zoals in proces besturingen, zitten bijna alle programma's in ROM. In die gevallen moet de industriële gebruiker zich beschermen tegen het wegvallen van de voeding; de programma's mogen niet vluchtig zijn. Ze moeten daarom in ROM staan.

Daar waar echter programma's vaak zullen worden gewijzigd, bij de hobbyist of in een omgeving waar programma's worden ontwikkeld, zullen deze in RAM staan. Later kunnen de programma's als ze foutloos zijn in ROM geplaatst worden. RAM is een vluchtig geheugen, d.w.z. wanneer de voeding uitvalt, of wordt uitgeschakeld, verdwijnt de inhoud.

De RAM (Random Access Memory) is een geheugen waarin gelezen en geschreven kan worden. In controle systemen zal de hoeveelheid RAM klein zijn (alleen voor data opslag). Daar waar echter programma's worden ontwikkeld zal de hoeveelheid RAM groot zijn. De RAM bevat het programma plus de ontwikkelings software. De RAM wordt vooraf geladen vanaf een extern randapparaat.

Tenslotte bevat het systeem een of meer interface chips, waarmee het systeem kan communiceren met de buitenwereld. De meest voorkomende interface chip is de PIO, de parallelle input/output chip. De PIO is, net als alle andere chips, verbonden met de drie bussen en voorziet het systeem van tenminste twee 8 bits poorten voor communicatie met de buitenwereld. Meer details over de werking van PIO's zijn te vinden in het boek "Microprocessor interface technieken". Details over het Z80 systeem staan in hoofdstuk 7 (Input/output apparatuur).

Alle chips zijn verbonden met de drie bussen van het systeem, maar voor de duidelijkheid zijn niet alle verbindingen getekend in de figuur.

Het is niet noodzakelijk dat iedere net besproken module op een eigen chip zit. We kunnen ook gecombineerde chips gebruiken, zoals een PIO gecombineerd met een kleine hoeveelheid ROM of RAM.

Om een echt systeem te bouwen zijn nog meer componenten nodig. Zo moeten de bussen gewoonlijk worden *gebufferd* en is er voor de RAM chips *decodeer logica* nodig en tenslotte zullen sommige signalen moeten worden versterkt door *drivers*. Deze hulpcircuits worden hier niet besproken, omdat ze voor het programmeren niet belangrijk zijn. De lezer die daarin is geïnteresseerd, wordt verwezen naar "Microprocessor interface technieken".

BINNEN IN EEN MICROPROCESSOR

De meeste microprocessors die nu op de markt zijn hebben de zelfde architectuur. Deze standaard architectuur wordt hier beschreven. Zie figuur 2.2. We zullen de verschillende modules bespreken die in deze figuur voorkomen van rechts naar links.

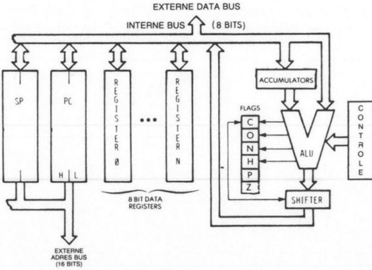


Fig. 2.2: "Standaard" microprocessor architectuur

Het hokje rechts met *controle* erin stelt de controle eenheid voor, welke zorg draagt voor de synchronisatie van het totale systeem. De rol die deze eenheid speelt zal in dit hoofdstuk worden uitgelegd.

De ALU voert rekenkundige en logische bewerkingen uit. Een speciaal register, de accumulator, vormt een van de ingangen van de ALU. (Er kunnen ook meerdere accumulators voorkomen.) In een instructie kan de accumulator zowel als ingang als uitgang (bron en bestemming) voorkomen.

De ALU kan ook *verschuiven* (*shift*) en *roteren* (*rotate*).

Tijdens een shift operatie wordt de inhoud van een byte een of meerdere posities naar links of naar rechts geschoven. Zie figuur 2.3. Ieder bit wordt een positie naar links geschoven. Details van deze bewerkingen komen in het volgende hoofdstuk aan bod.

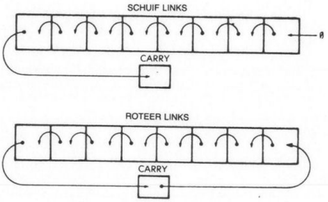


Fig. 2.3: Shift en rotate

Het circuit dat zorgt voor het verschuiven (de shifter) kan aan de uitgang van de ALU zitten, zoals in figuur 2.3, of aan de ingang van de accumulator.

Links van de ALU bevindt zich het *vlaggen* of *status register*. In dit register worden uitzonderlijke condities in de microprocessor bewaard. De inhoud kan getest worden door speciale instructies, of kan gelezen worden op de interne data bus. Een *voorwaardelijke* instructie veroorzaakt het uitvoeren van een nieuw programma, afhankelijk van de inhoud van een van deze bits.

De status bits komen later in dit hoofdstuk nog aan de orde.

Het veranderen van de vlaggen

De meeste instructies zullen de inhoud van een of meerdere vlaggen veranderen. Houd daarom altijd de kaart van de fabrikant bij de hand waarop staat welke instructies welke vlaggen veranderen. Alleen dan kan de juiste werking van een programma worden begrepen. In de bijlage is zo'n kaart voor de Z80 te vinden.

De registers

Kijk weer naar figuur 2.2. Links in deze figuur staan de registers van de microprocessor. Er kan onderscheid worden gemaakt tussen *algemene registers* en *adres registers*.

De algemene registers

D.m.v. de algemene registers kan de ALU data met hoge snelheid verwerken. Vanwege het beperkte aantal bits in een instructie is het aantal (direct te adresseren) registers meestal niet groter dan acht. Elk register bestaat uit een aantal flip-flops, die verbonden zijn met de tweerichtings interne data bus. De acht bits in een register kunnen tegelijk van of naar de data bus worden getransporteerd. Deze registers zijn uitgevoerd in MOS techniek en vormen het snelste geheugen op dit moment leverbaar. Hun inhoud is beschikbaar in enkele tientallen nanoseconden.

Interne registers zijn gewoonlijk genummerd van 0 tot n. De rol van deze registers ligt niet van te voren vast: ze zijn dus algemeen te gebruiken. Ze kunnen alle door het programma gebruikte data bevatten.

Normaal bevatten deze registers 8 bits data. In sommige microprocessors kunnen *twee* van deze registers tegelijk worden verwerkt. Deze registers heten dan samen "register paar". Hiermee kunnen 16 bits data of adressen worden opgeslagen.

De adres registers

Adres registers zijn 16 bits groot en dienen om adressen op te slaan. Ze worden vaak *data tellers* of *pointers* genoemd. Het zijn dubbele registers, d.w.z. twee 8 bits registers. Hun belangrijkste eigenschap is, dat ze verbonden zijn met de adres bus. De adres bus is beneden in figuur 2.4 te vinden.

De enige manier om de inhoud van deze 16 bits registers te laden is

via de data bus. Dit laden gebeurt in twee keer, de data bus kan immers slechts 8 bits bevatten. Om onderscheid te maken tussen deze twee bytes heten ze L (laag) en H (hoog) en bevatten respectivelijk de bits 0 tot en met 7, en bits 8 tot en met 15. In de meeste microprocessors zijn ten minste twee adres registers te vinden. "MUX" in figuur 2.4 betekent multiplexer.

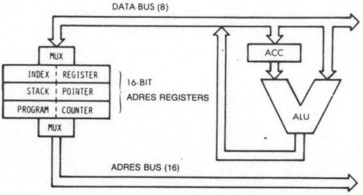


Fig. 2.4: De 16 bits adres registers maken de adres bus

Programma teller (PC)

De *programma teller* (engels: program counter) moet in iedere processor aanwezig zijn. Deze teller bevat het adres van de volgende uit te voeren instructie. De programmateller is onmisbaar en is de basis van de uitvoering van het programma. Dit alles zal in het volgende deel worden beschreven. Normaal wordt het programma sequentieel doorlopen. Om de volgende instructie uit te voeren is het nodig, dat deze naar de microprocessor wordt gehaald. De inhoud van de PC wordt op de adres bus gezet en naar het geheugen getransporteerd. De inhoud van het gespecificeerde adres in het geheugen wordt daarna terug gestuurd naar de MPU. Dit is de instructie. Enkele uitzonderlijke microprocessors, zoals de uit twee chips bestaande F8, bezitten geen programma teller. Dit betekent niet, dat het systeem geen PC heeft. Deze zit, om zo efficient mogelijk te kunnen werken, op de geheugen chip.

Stack pointer (SP)

De *stapel* (of *stack*) zal nu worden geintroduceerd. In de meeste krachtige general-purpose microprocessors (d.w.z. microprocessors die algemeen toepasbaar zijn) wordt de stapel door de "software" bepaald. Een deel van het geheugen wordt gereserveerd voor de stapel. Een speciaal 16 bits register houdt bij waar de top van de stapel zich bevindt. Dit register is de *stack pointer* of SP. De SP bevat het adres van de top van de stapel in het geheugen. Er zal nog worden aangetoond, dat de stapel onmisbaar is voor interrupts en subroutines.

Index register (IX)

Indexering is een manier om het geheugen te adresseren, die niet in iedere microprocessor voorkomt. De verschillende adresserings technieken worden beschreven in hoofdstuk 5. D.m.v. indexering kunnen blokken data met een enkele instructie worden geadresseerd. Een *index register* bevat een verplaatsing die automatisch wordt opgeteld bij de basis (of bevat een basis die opgeteld wordt bij een verplaatsing). Indexering wordt dus gebruikt om een woord in een blok instructies te bereiken.

De stapel

Formeel heet de stapel een LIFO structuur (LIFO = Last in, first out). Een stapel is een aantal registers, of geheugen locaties gereserveerd voor deze structuur. De chronologische structuur is karakteristiek voor de stapel. Het eerste element van de stapel komt altijd op de bodem daarvan terecht. Het laatst toegevoegde element vormt de top van de stapel. De stapel lijkt op een stapel serveerbladen in een zelfbedienings restaurant. De bladen worden opgestapeld in een gat met een veer op de bodem. Wordt een blad toegevoegd aan de stapel, dan komt deze bovenop te liggen. Wordt er een weggenomen, dan is dit de bovenste. Het oudste blad ligt dus onderop en het nieuwste blad ligt bovenop. Dit voorbeeld illustreert een andere karakteristiek van de stapel. Normaal is een element van de stapel alleen te bereiken via twee instructies: "push" en "pop". *Push* betekent dat een element aan de stapel wordt toegevoegd (bovenop). *Pop* haalt het bovenste element van de stapel. Bij de microprocessor wordt de inhoud van de *accumulator* op de top van de stapel geplaatst door een push. D.m.v. pop verhuist de top van de stapel naar de accumulator. M.b.v. andere speciale

instructies is het mogelijk het zelfde te doen met andere registers, zoals het status register. Wat dit betreft is de Z80 veelzijdiger dan de meeste andere microprocessors.

Voor subroutines, interrupts en tijdelijke data opslag is een stapel noodzakelijk. De rol van de stapel tijdens subroutines wordt verklaard in hoofdstuk 3 (Basis programmeer technieken). De rol die de stapel bij interrupts speelt komt aan de orde in hoofdstuk 6 (Input/output technieken). Hoe tijd gespaard kan worden bij tijdelijke data opslag m.b.v de stapel wordt verteld bij de programma's van de toepassingen.

Terwille van de eenvoud nemen we nu aan, dat een stapel noodzakelijk is in ieder computer systeem. Een stapel kan op twee manieren worden uitgevoerd:

1. Een vast aantal registers wordt hiervoor gereserveerd in de microprocessor. Dit is de "hardware stapel". Het voordeel is de hoge snelheid. Een nadeel is het beperkte aantal registers.

2. De meeste general-purpose microprocessors hebben een andere benadering van de stapel, de "software stapel". Zo ook in de Z80. In dit geval bevat register SP de "stack pointer", dat is het adres van de top van de stapel (of in sommige gevallen het adres plus 1). De stapel is nu een gebied in het geheugen. De SP is 16 bits groot om ieder adres in het geheugen aan te kunnen wijzen.

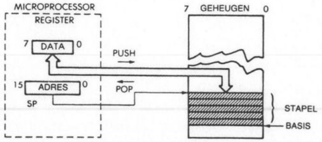


Fig. 2.5: De stapel manipulatie instructies

De instructie executie cyclus

Bekijk nu figuur 2.6. De MPU zit links en het geheugen rechts. De geheugen chip kan ROM of RAM zijn, of iedere andere chip die ge-

heugen bezit. Het geheugen wordt gebruikt om instructies en data op te slaan. We willen hier een instructie uit het geheugen halen om de rol van de programma teller te illustreren. Neem aan dat de PC een geldige inhoud heeft. Dit is een 16 bits adres van de volgende uit het geheugen te halen instructie. Iedere processor gaat nu door met de volgende drie cycli: ·

1 - haal de volgende instructie (dit heet de "fetch")
2 - decodeer de instructie
3 - voer de instructie uit

Fetch

We nemen nu de volgende reeks gebeurtenissen onder de loep. Tijdens de eerste cyclus wordt de inhoud van de programma teller op de adres bus gezet en naar het geheugen gestuurd. Tegelijker tijd wordt er, indien dat noodzakelijk is, een lees signaal op de controle bus gegenereerd. Het geheugen ontvangt het adres. Het adres wordt gebruikt om een bepaalde geheugen locatie aan te wijzen. Als het geheugen het adres ontvangen heeft, wordt het door interne decoders gedecodeerd, en het betreffende adres wordt geselecteerd. Een paar honderd nanoseconden later plaatst het geheugen de inhoud van de geselecteerde geheugen locatie op de data bus. Dit 8 bits grote woord is de instructie die we wilden ophalen.

Korte samenvatting: de inhoud van de programma teller wordt op de adres bus gezet. Er wordt een lees signaal gegenereerd. Het geheugen verwerkt dit (dit heet een geheugen cyclus), en ongeveer 300 nanoseconden later wordt de gewenste instructie op de data bus gezet. (Aangenomen dat het een een-bytes instructie was). De microprocessor leest de data bus en plaatst de inhoud daarvan in een speciaal register, het IR register. Het IR is het *instructie register*, het is acht bits groot en bevat de laatste uit het geheugen gehaalde instructie. Het IR verschijnt links in figuur 2.7. Het IR is niet toegankelijk voor de programmeur.

Decodering en executie

Wanneer de instructie eenmaal in het IR is, kan de controle eenheid van de microprocessor de inhoud ervan decoderen en dan de juiste volgorde van de diverse interne en externe signalen generen om de instructie uit te voeren. Daarom is er, voordat de instructie uitgevoerd kan worden, een vertraging, waarvan de tijdsduur afhangt van de soort

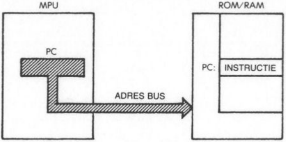


Fig. 2.6: Het ophalen van een instructie uit het geheugen

instructie. Sommige instructies worden totaal binnen de microprocessor uitgevoerd. Bij andere instructies wordt data van of naar het geheugen getransporteerd. Daarom duurt het uitvoeren van de verschillende instructies niet even lang. De tijdsduur wordt uitgedrukt in het aantal benodigde (klok) cycli. Zie voor deze aantallen de bijlage.

Het opgeven van de tijdsduur in nanoseconden is niet zinvol, omdat de klokfrequentie variabel is.

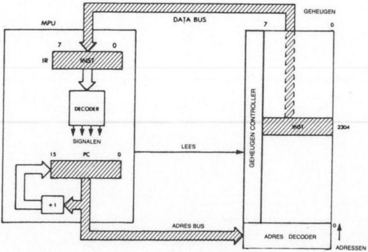


Fig. 2.7: Automatische sequencing

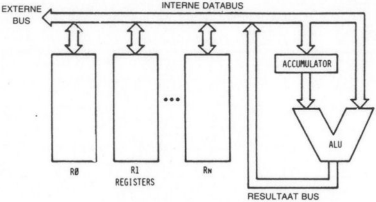


Fig. 2.8: Een-bus architectuur

Het ophalen van de volgende instructie

We hebben beschreven hoe m.b.v. de programma teller een instructie uit het geheugen wordt gehaald. Tijdens de uitvoering (executie) van het programma worden de instructies in een opeenvolgende reeks opgehaald uit het geheugen. Daarom moet er voorzien worden in een automatisch mechanisme dat dit doet. Deze taak wordt uitgevoerd door een eenvoudige opteller die verbonden is met de programma teller. Dit is in figuur 2.7 geillustreerd. Iedere keer als de inhoud van de programma teller op de adres bus wordt geplaatst wordt de programma teller met 1 verhoogd. Als, bijvoorbeeld, de inhoud van de PC "0" is. wordt "0" op de adres bus gezet. Daarna wordt de inhoud met "1" verhoogd, en terug gezet in de programma teller. Deze wordt dus nu "1". In de volgende cyclus wordt dus de instructie op adres "1" opgehaald. Dit mechanisme wordt, in goed nederlands, *automatische sequensing* genoemd.

Ik moet er op wijzen dat deze beschrijving een vereenvoudigde weergave van de werkelijkheid is. Instructies kunnen tot drie bytes lang zijn, zodat meerdere bytes uit het geheugen moeten worden gehaald. Het principe van het mechanisme blijft echter gelijk. De programma teller wordt gebruikt om zowel opeenvolgende bytes van een instructie op te halen, als opeenvolgende instructies. De programma teller samen met de opteller vormen een automatisch mechanisme dat steeds wijst naar de volgende op te halen byte.

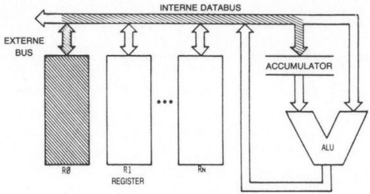


Fig. 2.9: Executie van een optelling-RO in ACC

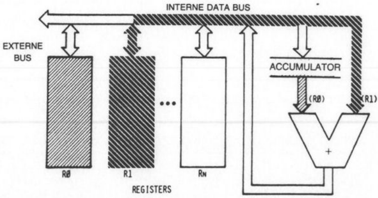


Fig. 2.10: Optelling-tweede register R1 in ALU

We zullen nu een instructie binnen de MPU uitvoeren (zie figuur 2.8). Een typische instructie zou zijn: R0 = R0 + R1. Dit betekent:"tel de inhouden van R0 en R1 bij elkaar op en plaats het resultaat in R0". Om deze operatie uit te voeren, wordt de inhoud van R0 gelezen en via de bus op de linker ingang van de ALU geplaatst, in het daar aanwezige buffer register. Daarna wordt R1 geselecteerd en de inhoud daarvan komt via de bus op de rechter ingang van de ALU te staan. Dit is geillustreerd in de figuren 2.9 en 2.10. Op dit punt staat op de rechter ingang van de ALU de inhoud van R1 en op de linker ingang de inhoud van het buffer register, de voorgaande waarde van R0. De optelling kan nu worden uitgevoerd. Dit gebeurt door de ALU, en het resultaat verschijnt op de uitgang ervan, rechts-onder in figuur 2.11. Het resultaat wordt via de bus naar R0 getransporteerd. De executie van de instructie is voltooid. Het resultaat van de optelling staat in R0. Merk op dat de inhoud van R1 niet door deze instructie is veranderd. Dat is trouwens een algemeen geldend principe: de inhoud van een register of van een lees/schrijf geheugen wordt niet veranderd door een lees operatie.

Het buffer register was nodig om de inhoud van R0 te onthouden, zodat de bus gebruikt kon worden voor een ander transport. Er blijft echter nog een probleem over.

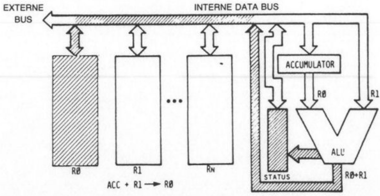


Fig. 2.11: Het resultaat wordt gegenereerd en gaat naar R0

De kritieke race

De eenvoudige organisatie zoals die te zien is in figuur 2.8 zal niet op de juiste wijze werken.

Vraag: Wat is het timing probleem?

Antwoord: Het probleem is, dat het resultaat op de uitgang van de ALU terecht komt op de bus. Het wordt niet alleen getransporteerd in de richting van R0, maar komt overal op de bus beschikbaar. Het resultaat komt dus ook op de rechter ingang van de ALU te staan! Het gevolg is, de uitgang een paar nanoseconden later weer verandert. Dit is een *kritieke race*. De uitgang van de ALU moet geisoleerd worden van zijn ingang (Zie figuur 2.12).

Meerdere oplossingen zijn mogelijk om dit te doen. Een buffer register moet worden gebruikt. De buffer kan aan de ingang of aan de uitgang worden geplaatst. Meestal aan de ingang. Hier is het geplaatst aan de rechter ingang. Door het bufferen van het systeem werk het nu op de juiste manier. Later in dit hoofdstuk zal worden aangetoond, dat als het !inker register als accumulator wordt gebruikt, deze ook een buffer nodig heeft. Dit is te zien in figuur 2.13.

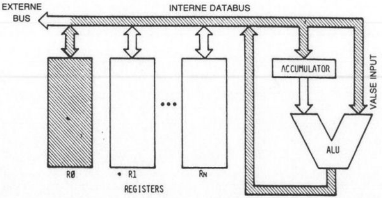


Fig. 2.12: De kritieke race

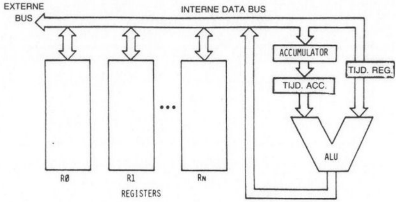


Fig. 2.13: Twee buffers zijn nodig

INTERNE ORGANISATIE VAN DE Z80

De namen die nodig zijn om de interne elementen van de microprocessor te begrijpen zijn gedefinieerd. We kunnen nu meer gedetaillerd de Z80 aan een onderzoek onderwerpen, en de capaciteiten ervan beschrijven. De interne organisatie van de Z80 is te vinden in figuur 2.14. Dit diagram geeft de logische opbouw van het apparaat weer. Er kunnen nog meerdere onderlinge verbindingen voorkomen, maar deze staan niet in de figuur. We bekijken de figuur van rechts naar links.

Rechts bevindt zich de *rekenkundige-logische eenheid* (ALU), te herkennen aan de karakteristieke "V" vorm. De accumulator, beschreven in de vorige sectie en gekenmerkt als A, is te vinden bij de rechter ingang van de ALU. De accumulator is uitgerust met een *buffer* ACT (tijdelijke accumulator). De linker ingang van de ALU heeft eveneens een *tijdelijke buffer*. Deze heeft de naam TMP meegekregen. De werking van de ALU zal in de volgende sectie duidelijk worden als enkele instructies worden uitgelegd.

Het *vlag register* heet in de Z80 "F" (F van flag) en staat rechts van de accumulator. De inhoud van dit register wordt voornamelijk bepaald door de ALU, maar we zullen later zien dat ook andere modules of gebeurtenissen invloed daarop kunnen hebben.

De accumulator en het vlag register staan afgebeeld als dubbele re-

gisters A, A' en F, F'. Dit vanwege het feit, dat de Z80 intern is uitgerust met twee sets registers: A + F, en A' + F'. Er mag echter slechts een set tegelijk gebruikt worden. Er is een instructie, waarmee de inhouden van A en F verwisseld kunnen worden met die van A' en F'. Om het geheel eenvoudig te houden staan in de meeste nu volgende diagrammen alleen A en F afgebeeld. De lezer moet onthouden, dat hij de mogelijkheid heeft over te schakelen naar het alternatieve register paar.

De rol van iedere vlag wordt uitgelegd in hoofdstuk 3 (Basis programmeer technieken).

Een groot blok met registers staat in het midden van de tekening. De bovenste registers zijn dubbel uitgevoerd. Dit zijn de registers B,C,D,E,H en L. Dit zijn de algemene of *general-purpose acht bits registers* van de Z80. De Z80 heeft twee eigenaardigheden in vergelijking met de in het begin van dit hoofdstuk genoemde microprocessor.

Ten eerste heeft de Z80 twee *register banken*, d.w.z twee identieke groepen van 6 registers elk. Slechts 6 registers kunnen tegelijk gebruikt worden. Er zijn echter speciale instructies waarmee tussen de twee banken geschakeld kan worden. Een bank gedraagt zich dus als een intern geheugen, terwijl de andere bank de werkregisters vormt. Het mogelijke gebruik ervan wordt in het volgende hoofdstuk beschreven.

Om verwarring te voorkomen bekijken we tijdelijk alleen de werkregisters en negeren we de tweede bank.

MUX boven de geheugen bank is een afkorting van *multiplexer*. De data komende van de interne data bus worden door de multiplexer naar het juiste register gestuurd. Er kan slechts een register tegelijkertijd met de data bus verbonden zijn.

Een tweede karakteristiek van deze zes registers is, buiten het feit dat het general-purpose registers zijn, dat ze voorzien zijn van een verbinding met de *adres bus*. Daarom zijn ze ook in *paren* gegroepeerd. Zo kunnen bijvoorbeeld de inhouden van B en C tegelijk op de adres bus worden gezet. Op deze wijze kan deze groep registers niet alleen worden gebruikt om acht bits data op te slaan, maar ook kunnen ze dienen als 16 bits *pointers* voor geheugen adressering.

De derde groep registers, te vinden onder de vorige groepen, zijn echte adres registers. In totaal bevat de groep vier registers. Als in iedere microprocessor vinden we er de programma teller (PC) en de stack pointer (SP). Herinner je dat de programma teller het adres van de volgende uit te voeren instructie bevat.

De stack pointer wijst naar de top van de stapel. Bij de Z80 is dit het adres van de laatste bezette plaats. (Bij andere microprocessors wijst

de pointer naar de eerste vrije plaats.) De stapel groeit naar beneden, d.w.z. in de richting van de lagere adressen.

Dit betekent, dat bij iedere push de pointer wordt verlaagd. Omgekeerd wordt de teller verhoogd bij iedere pop. Omdat iedere push en pop betrekking heeft op twee woorden, wordt de stack pointer verhoogd en verlaagd met twee.

In deze groep registers vinden we een nieuw, nog niet eerder beschreven, soort register: het *index register*. Dit zijn de registers IX (Index register X), en IY (Index register Y). De index registers zijn uitgerust met een speciale opteller, in figuur 2.14 te zien als een miniatuur ALU. Een byte op de data bus kan hierdoor worden opgeteld bij de inhoud van IX of IY. Deze byte heet de *verplaatsing* bij een geindexeerde instructie. Er zijn instructies, die automatisch de inhoud van IX of IY optellen bij de verplaatsing en hiermee een nieuw adres maken. Dit wordt indexeren genoemd. Het is een gemakkelijke manier om toegang te krijgen tot een opeenvolgende reeks data. In hoofdstuk 5 wordt dit besproken.

Tenslotte vinden we linksonder de registers een blok met "+/−1". M.b.v. dit blok kan de inhoud van de pure adres registers, iedere keer als een adres op de bus wordt gezet, met 1 worden verhoogd of verlaagd. Dit is zeer belangrijk voor het maken van "programmalussen", zoals besproken zal worden in de volgende sectie. Deze faciliteit maakt de toegang tot opeenvolgende geheugen locaties erg gemakkelijk.

We gaan nu naar de linker kant van de tekening. Daar is een register paar te zien: I en R. I heet het *interrupt-pagina adres register*. I wordt besproken in hoofdstuk 6 (input/output technieken). Het wordt alleen gebruikt in een speciale mode, als een indirecte "call" wordt gegenereerd als reactie op een interrupt. In het I register wordt het hoge orde deel van het indirecte adres bewaard. Het lage orde deel wordt gegenereerd door het apparaat dat de interrupt veroorzaakt.

Het R register is het *geheugen-refresh register*. Dit register wordt gebruikt voor de automatische refresh van dynamische geheugens. Gewoonlijk wordt dit register buiten de microprocessor aangetroffen, daar het eigenlijk een onderdeel van het geheugen is. Dat dit register nu in de Z80 zit is erg gemakkelijk, want het beperkt de hoeveelheid benodigde hardware bij gebruik van dynamische geheugens. Voor het programmeren is dit register niet van belang. (Voor een gedetailleerde beschrijving van geheugen-refresh technieken raadplege men "Microprocessor interface technieken"). Het is, bijvoorbeeld, mogelijk het register te gebruiken als een software klok.

Helemaal links in de tekening vinden we het controle blok van de

microprocessor. Bovenaan vinden we het *instructie register* IR, dat de uit te voeren instructie bevat. Het IR register is totaal verschillend van het eerder beschreven "I,R" register paar. De instructie wordt uit het geheugen gehaald, getransporteerd over de interne data bus en komt tenslotte terecht in het instructie register. Onder het instructie register bevindt zich de *decoder*, die signalen naar de controller-sequencer zendt en zorg draagt voor de uitvoering van de instructie binnen en buiten de microprocessor. Het *controle blok* genereert de signalen op en heeft het beheer over de controle bus onder in de tekening.

De drie bussen van het systeem, d.w.z de data bus, de adres bus en de controle bus, worden buiten de microprocessor voort gezet en zijn bereikbaar via de pootjes van de chip. Deze externe verbindingen zijn rechts in de tekening te vinden. De bussen zijn van de buitenwereld geïsoleerd d.m.v. buffers. (zie figuur 2.14).

Alle logische elementen van de Z80 zijn nu beschreven. Om te kunnen beginnen met het schrijven van programma's is het niet noodzakelijk de werking van de Z80 tot in details te kennen. Om echter efficiente programma's te ontwerpen moet men een juiste keuze kunnen maken tussen de verschillende registers en technieken. Daarvoor is een goed begrip van de werking van de microprocessor nodig. Daarom zullen we nu de executie van typische instructies in de Z80 onderzoeken en de rol en het gebruik van de interne registers en bussen demonstreren.

INSTRUCTIE FORMATEN

De Z80 instructies zijn te vinden in de bijlage. Z80 instructies zijn opgebouwd uit een, twee, drie of vier bytes. Een instructie bepaalt de handelingen die door de microprocessor worden uitgevoerd. We kunnen, om het even eenvoudig te houden, zeggen, dat een instructie bestaat uit een operatie code (opcode) en eventueel een letter of adres veld, bestaande uit een of twee woorden. De opcode bepaalt de uit te voeren operatie. Of in de formele computer terminologie: de opcode representeert alleen die bits die de uit te voeren operatie specificeren, exclusief de eventueel aan te wijzen registers. In de computer wereld wordt echter uit oogpunt van gemak de operatie plus de bijbehorende registers opcode genoemd. Voor een efficiente werking moet de "gegeneraliseerde opcode" in een byte passen (dit is de beperkende factor van het aantal beschikbare instructies in een microprocessor).

De 8080 heeft een, twee of drie bytes lange instructies (zie figuur

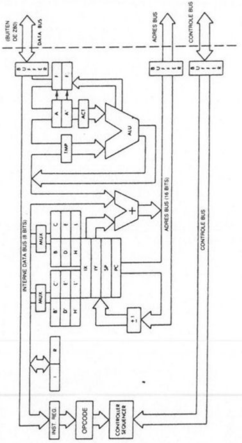


Fig. 2.14: Interne Z80 organisatie

2.15). De Z80 is bovendien nog uitgerust met geindexeerde instructies, die een byte meer nodig hebben. De opcodes zijn in het algemeen bij de Z80 een byte lang, uitgezonderd enkele speciale instructies, die twee bytes lange opcodes hebben.

Bij sommige instructies wordt de opcode gevold door een data byte. In dit geval is de instructie een twee bytes instructie (behalve bij indexering, waar een extra byte nodig is).

Andere instructies specificeren een 16 bits adres. De instructie is dan drie of vier bytes lang.

Iedere byte van de instructie moet uit het geheugen worden gehaald, wat iedere keer vier klokcycli duurt. Hoe korter de instructie derhalve is, hoe korter de tijd is om deze uit te voeren.

Een een-woord instructie

In principe zijn de een-woord instructies het snelst, en zijn daarom favoriet bij de programmeur. Voor de Z80 is een voorbeeld van zo'n instructie:

LD r,r'

Deze instructie betekent:"plaats de inhoud van register r' in r". Het is een typische "register naar register" operatie. Iedere microprocessor moet voorzien zijn van dergelijke instructies, waarmee data van het ene interne register naar het andere wordt verplaatst. Instructies die betrekking hebben op speciale machine registers, zoals de accumulator, kunnen een speciale opcode hebben.

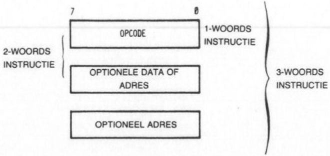


Fig. 2.15: Typische instructie formaten

Na de uitvoering van de instructie LD r,r' is de inhoud van r gelijk aan die van r'. De inhoud van r' is niet veranderd door de lees operatie.

Iedere instructie wordt intern gerepresenteerd in een binair formaat. De notatie LD r,r' heet symbolisch of mnemonisch. Het wordt de machinetaal voorstelling van de instructie genoemd. Dit is makkelijker leesbaar dan de binaire voorstelling. De binaire code van deze instructie in het geheugen is: 01DDDSSS (bits 0 tot en met 7).

Deze voorstelling is gedeeltelijk symbolisch. Iedere D en S staat voor een binair bit. De drie D's wijzen naar het register waar de data naar toe moet (*destination register*). Omdat drie bits worden gebruikt, kunnen we kiezen uit acht registers. De codes van de registers staan in figuur 2.16. Register B heeft bijvoorbeeld de code "000", en C de code "001", enz.

Op de zelfde wijze wijst "SSS" naar het register waar de data vandaan komt (*source register*). De afspraak is, dat r' de bron is en r de bestemming. De plaatsing van de bits in de code is niet voor het gemak van de programmeur, maar voor de controle sectie, die de instructie moet decoderen en uitvoeren. Daar tegenover staat, dat de machine taal bedoeld is om het gemak van de programmeur te dienen. Dat LD r,r' niet betekent: "verplaats de inhoud van r naar r'" is uitsluitend gedaan om een beetje in overeenstemming te blijven met de binaire code. Het is natuurlijk een arbitraire beslissing.

Oefening 2.1: Schrijf de binaire code op voor de instructie: "Verplaats de inhoud van register C naar register B". Raadpleeg figuur 2.16 voor de codes voor B en C.

Een ander voorbeeld voor een een-woord instructie is:

ADD A,r

Deze instructie telt de inhoud van een gespecificeerd register (r) op bij de accumulator (A). Het resultaat komt in A. We kunnen dit ook opschrijven als: A = A + r. In de bijlage is te vinden dat de binaire voorstelling voor deze instructie is:

10000SSS

SSS is het register, dat bij de accumulator wodt opgeteld. De zelfde codes voor de registers als bij LD r,r' zijn hier geldig.

Oefening 2.2: Wat is de binaire code voor de instructie die de inhoud van register D bij de accumulator opteld?

CODE	REGISTER
0 0 0	B
0 0 1	C
0 1 0	D
0 1 1	E
1 0 0	H
1 0 1	L
1 1 0	-GEHEUGEN
1 1 1	A

Fig. 2.16: De register codes

Een twee-woords instructie

ADD A,n

Deze eenvoudige instructie telt de inhoud van de tweede byte van de instructie op bij die van de accumulator. De inhoud van de instructie heet een "literal". Het is gewone data en wordt zonder enige verdere betekenis behandeld. Het kan een karakter of numerieke data zijn. Dit is onbelangrijk voor de operatie. De code voor de instructie is:

11000110 gevolgd door de 8 bits byte "n"

Het is een zogenaamde *immediate* operatie. "Immediate" betekent in de meeste programmeer talen, dat het volgende woord, of woorden in de instructie een stukje data is, dat niet op de manier van een opcode geinterpreteerd mag worden. Deze woorden moeten opgevat worden als een *literal*.

De controle eenheid weet hoeveel woorden iedere instructie heeft. Daarom zal het altijd het juiste aantal woorden ophalen uit het geheugen en uitvoeren. Hoe langer echter de instructies worden, des te moeilijker het voor de controle eenheid wordt ze te decoderen.

Een drie-woords instructie

LD A,(nn)

De instructie gebruikt drie woorden. Het betekent: "Laadt de accumulator met de inhoud van het door de laatste twee bytes van de instructie gespecificeerde geheugen adres". Omdat adressen 16 bits lang zijn, hebben we daarvoor twee woorden nodig. De instructie binair weergegeven is:

0 0 1 1 1 0 1 0:	8 bits voor de opcode
laag adres:	8 bits van het lagere orde deel v.h. adres
hoog adres:	8 bits van het hogere orde deel v.h. adres

HET UITVOEREN VAN INSTRUCTIES BINNEN DE Z80

We hebben gezien dat alle instructies in drie fases worden uitgevoerd: FETCH, DECODEER en EXECUTEER. Er moeten nu enkele definities worden geintroduceerd. Iedere fase heeft enkele klokcycli nodig. De Z80 executeert iedere fase in een of meer logische cycli, die "machine cycli" worden genoemd. De kortste machine cyclus duurt drie klokcycli.

Toegang krijgen tot een geheugen duurt vier klokcycli. De meeste instructies hebben er meer nodig.

Iedere machine cyclus heeft een naam: M1, M2, enz. Dit geldt ook voor de klokcycli: T1, T2, enz.

De FETCH fase

De FETCH fase van de instructie bestaat uit de eerste drie klokcycli van de machine cyclus M1: T1, T2 en T3. Alle instructies hebben deze fase gemeen, want alle instructies moeten uit het geheugen worden gehaald voordat ze gedecodeerd kunnen worden. Het FETCH mechanisme werkt als volgt:

T1: PC UIT

Allereerst moet het adres van de volgende instructie op de adres bus worden gezet. Dit adres bevindt zich in de programma teller. (Zie fi-

guur 2.17). Het geheugen ontvangt het adres, en decodeert het om de juiste geheugen locatie te selecteren. Enkele honderden nanoseconden zullen verstrijken (een nanoseconde is 10^{-9} seconde) voordat de inhoud van het geheugen op de uitgang daarvan beschikbaar is.

De uitgang is met de data bus verbonden. In het standaard microprocessor ontwerp wordt de leestijd van het geheugen gebruikt voor een operatie binnen de microprocessor. Deze operatie is de verhoging van de programma teller met 1:

T2: PC = PC + 1

Zie figuur 2.18. Aan het eind van T2 is de inhoud van het geheugen beschikbaar en kan naar de microprocessor worden getransporteerd:

T3: INST naar IR

De DECODEER en EXECUTIE fases

Gedurende T3 is de instructie uit het geheugen gelezen en via de data bus getransporteerd naar het instructie register, waar het zal worden gedecodeerd.

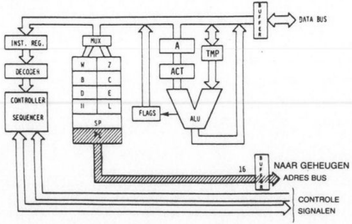


Fig. 2.17: Instructie fetch-(PC) wordt naar het geheugen gestuurd

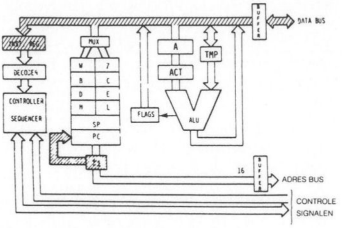


Fig. 2.18: PC wordt met 1 verhoogd

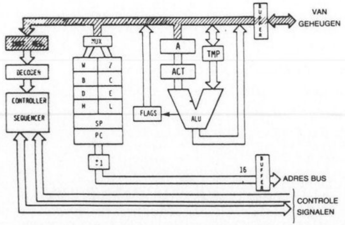


Fig. 2.19: De instructie wordt uit het geheugen gehaald en in het instructie register geladen.

Opgemerkt dient te worden, dat T4 van M1 altijd nodig is. Als de instructie immers in IR geladen is gedurende T3, moet deze (de instructie) gedecodeerd en uitgevoerd worden. Daar is tenminste 1 klokcyclus voor nodig: T4.

Een paar instructies hebben nog een extra cyclus nodig, T5. De meeste instructies hebben deze echter niet nodig. Als een instructie meer machine cycli nodig heeft, dan gaat T4 van M1 direct over in T1 van M2. We bekijken een voorbeeld. Een gedetaillerd overzicht van de benodigde cycli is te vinden in figuur 2.27. Omdat deze tabel voor de Z80 nog niet is vrijgegeven, is die van de 8080 gebruikt. Deze tabellen zorgen voor een diepgaand begrip voor de wijze waarop instructies worden uitgevoerd.

LD D,C

Dit is de zelfde instructie als MOV r1,r2 voor de 8080. Zie regel 1 van figuur 2.27. Het destination register (daar waar de data naar toe moet) is D. De overdracht is geillustreerd in figuur 2.20.

Deze instructie is beschreven in de vorige sectie. Het verplaatst de inhoud van register C naar register D.

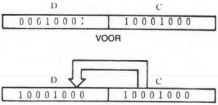


Fig. 2.20: Verplaatsing van C naar D

De eerste drie "states" (een state is een klokcyclus) van M1 worden gebruikt voor de fetch van de instructie uit het geheugen. Aan het einde van T3 is de instructie in het instructie register, waar het gedecodeerd kan worden. Zie figuur 2.19.

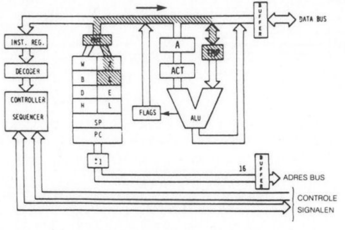


Fig. 2.21: De inhoud van C gaat naar TMP

Gedurende T4: (SSS) ► TMP

De inhoud van C komt in TMP (Zie figuur 2.21).

Gedurende T5: (TMP) ► DDD

De inhoud van TMP gaat naar D. Dit wordt getoond in figuur 2.22.

De instructie is nu voltooid. De inhoud van register C is verplaatst naar het gespecificeerde destination register D. Hiermee wordt de executie van de instructie beeindigd. De andere machine cycli M2, M3, M4 en M5 zijn niet nodig en er wordt dus gestopt na M1.

De tijdsduur van de instructie kan gemakkelijk worden berekend. Iedere state duurt immers 1 klokcyclus, en deze is 500 nanoseconden. De instructie duurt vijf states. De tijdsduur is dus: 5 X 500 = 2500 ns = 2,5 microseconden.

Vraag: Waarom zijn er twee states nodig om de inhoud van C naar D te verplaatsen in plaats van een? Het transport loopt via TMP. Is het niet gemakkelijker direct de inhoud te verplaatsen van C naar D, zodat er slechts een state nodig is?

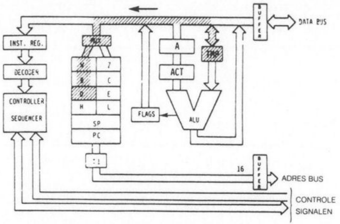


Fig. 2.22: De inhoud van TMP gaat naar D

Antwoord: Dit is niet mogelijk vanwege de onderlinge interne opbouw van de registers. Alle registers zijn in feite onderdelen van een enkele RAM, een lees/schrijf geheugen op de microprocessor chip. Er kan slechts een register tegelijk worden geadresseerd of geselecteerd. Daarom kan niet tegelijk worden gelezen en geschreven op twee verschillende geheugen locaties van de RAM. Er zijn twee RAM cycli nodig. We moeten dus eerst het register C lezen, en de inhoud daarvan opslaan in een tijdelijk geheugen. Daarna kan de data terug geschreven worden in register D. Dit is een onvolkomenheid in het ontwerp. Deze beperking is bijna gemeengoed bij vrijwel alle een-chips microprocessors. Een dubbele RAM is nodig om dit probleem op te lossen. De ontwerp beslissing hoeft zeer zeker niet voor alle microprocessors te gelden, en deze beperking komt normaal ook niet voor in bit-slice processors. De reden dat deze beperking voorkomt is het feit, dat de beschikbare ruimte op de chip zo efficient mogelijk gebruikt moet worden. Misschien wordt dit wel verbeterd in de toekomst.

Belangrijke oefening:

Nu we op dit punt zijn aangekomen is het raadzaam de oefening nog eens te herhalen. Ga terug naar figuur 2.14, en probeer m.b.v. kleine voorwerpen de door de data gevolgde weg te simuleren. Leg bijvoor-

beeld een voorwerp in PC. Tijdens T1 wordt dit voorwerp naar het geheugen getransporteerd. Ga op deze manier door totdat alles goed begrepen is. Pas daarna zijn we gereed om verder te lezen.

Steeds moeilijker instructies zullen nu worden bekeken:

ADD A,r

Deze instructie betekent: "tel de inhoud van register r op bij die van A (accumulator) en plaats het resultaat in A". Het is een *impliciete* instructie. De instructie heet impliciet, omdat een tweede register niet expliciet genoemd wordt. De instructie geeft expliciet alleen register r aan. Geimpliceerd wordt dat het andere register de accumulator is. In dergelijke instructies is de accumulator zowel bron als bestemming (source en destination). Het resultaat van de optelling komt in de accumulator terecht. Het voordeel van een impliciete instructie is, dat de opcode slechts 1 byte lang is. Er zijn maar drie bits nodig om register r te specificeren. Op deze manier wordt de optelling snel uitgevoerd.

Andere impliciete instructies bestaan in de Z80, met andere gespecialiseerde registers. Meer ingewikkelde voorbeelden hiervan zijn PUSH en POP, waarbij data wordt getransporteerd tussen de top van de stapel en de accumulator. Tegelijk wordt de stack pointer (SP) bijgewerkt. In dit geval is SP het geimpliceerde register.

We zullen nu de ADD A,r instructie in detail bestuderen. De instructie heeft twee machine cycli, M1 en M2, nodig. Zoals gebruikelijk is wordt gedurende de eerste drie states van M1 de instructie uit het geheugen gehaald en in het instructie register gezet. Aan het begin van T4 wordt de instructie gedecodeerd en kan deze worden uitgevoerd. We nemen aan dat register B bij de accumulator wordt opgeteld. De code voor de instructie is dan: 10000000 (de code voor register B is 000). Het 8080 equivalent voor de instructie is ADD r.

T4: (SSS) ► TMP, (A) ► ACT

Twee data verplaatsingen gebeuren gelijktijdig. Ten eerste wordt de inhoud van het source register B naar TMP gebracht, dat is de rechter ingang van de ALU (zie figuur 2.23). Tegelijk wordt de inhoud van de accumulator in de tijdelijke accumulator gezet (ACT). Door figuur 2.23 te bekijken kunnen we er ons van overtuigen, dat dit inderdaad mogelijk is. De transporten gebruiken twee verschillende datapaden in het systeem. De overdracht van B naar TMP gebruikt de interne data bus. De data van A naar ACT gebruikt een kort pad, dat onafhankelijk

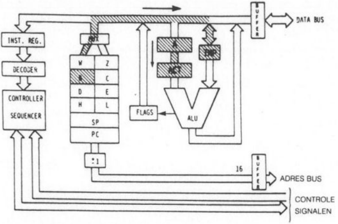


Fig. 2.23: Twee transporten gebeuren gelijktijdig

van de interne data bus is. Om tijd te sparen kunnen beide transporten dus gelijktijdig gebeuren. De juiste getallen staan nu op de ingangen van de ALU. We kunnen dus gaan optellen. Je zou verwachten dat dit zou gebeuren gedurende T5 van M1, maar niets is minder waar: T5 van M1 wordt niet gebruikt. We beginnen aan M2. Maar ook T1 van M2 wordt niet gebruikt! De optelling vindt plaats gedurende T2 van M2 (Zie ADD r in figuur 2.27).

T2 van M2: (ACT) + (TMP) ► A

Het resultaat van de optelling komt in de accumulator. Zie figuur 2.24. De optelling is klaar.

Vraag: Waarom wordt de beeindiging van de instructie uitgesteld tot T2 van M2, in plaats van T5 van M1? (Dit is een moeilijke vraag, waarvoor je het ontwerp van de CPU moet begrijpen. De techniek die hier echter is gebruikt is fundamenteel voor klok-synchrone CPU ontwerpen. Probeer daarom te begrijpen wat er gebeurt.)

Antwoord: Hier is een standaard truc gebruikt, die in bijna iedere microprocessor voorkomt. Deze truc heet de "fetch/executie overlapping". Het basis idee is het volgende: terug kijkend naar figuur 2.23 is

te zien, dat de optelling zelf alleen de data bus en de ALU gebruikt. In het bijzonder wordt geen gebruik gemaakt van de register RAM. Wij (en de controle unit) weten dat na beeindiging van de instructie de cycli T1, T2, en T3 van M1 van de volgende instructie zullen worden uitgevoerd. We weten ook, dat daarvoor alleen de data bus en de programma teller nodig zijn. Daarvoor moeten we toegang hebben tot de register RAM. (Daarom kunnen we deze truc niet uithalen bij LD r,r'). We kunnen dus tegelijk het gearceerde gebied van figuur 2.17 en het gearceerde gebied van figuur 2.24 gebruiken.

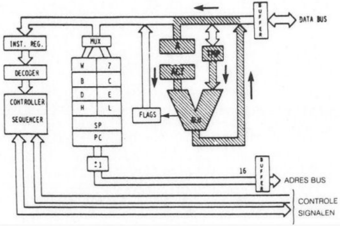


Fig. 2.24: Einde van ADD r

De data bus wordt tijdens T1 van M2 gebruikt om status informatie over te brengen. Deze kan dus niet voor de optelling worden gebruikt. We moeten dus tot T2 van M2 wachten voor we door kunnen gaan met de optelling. Het mechanisme is nu uitgelegd. De voordelen van deze benadering moeten duidelijk zijn. Stel dat de instructies recht-toe-recht-aan zouden worden uitgevoerd, dan was de optelling voltooid na T5 van M1. Dat zou 5 X 500 = 2500 ns duren. Met de nu geschetste methode wordt de nieuwe instructie al geinitieerd als de oude instructie met T4 bezig is. Op een manier die onzichtbaar is voor de nieuwe instructie wordt tijdens T2 de oude afgemaakt.

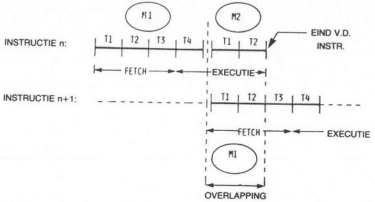


Fig. 2.25: Overlapping van FETCH-EXECUTIE tijdens T1-T2

In de tekening is T2 een onderdeel van M2. M2 is de tweede machine cyclus van de optelling. In feite is M2 identiek aan M1 van de volgende instructie. De vertraging geintroduceerd door de ADD instructie duurt voor de programmeur slechts 4 × 500 = 2000 ns, i.p.v. de 2500 ns van de recht-toe-recht-aan aanpak. Dat is dus 20% sneller!

De techniek is geillustreerd in figuur 2.25. Hij wordt gebruikt waar dat maar mogelijk is om de snelheid van het systeem op te voeren. Natuurlijk is deze techniek niet overal toe te passen. De controle eenheid weet wanneer een overlapping mogelijk is.

Vraag: Kan de gebruikte techniek ook toegepast worden op langere instructies, door T3 van M2 op dezelfde wijze te gebruiken?

Om daar achter te komen raad ik aan figuur 2.27 goed te bestuderen. Op deze manier kan men zich de interne werking van de microprocessor eigen maken. Deze tabellen gelden voor de 8080, maar de Z80 kent deze instructies ook (plus nog enkele meer). Voor de Z80 zijn deze gegevens niet beschikbaar. De overeenkomsten tussen de Z80 en de 8080 staan in de bijlage.

We gaan een meer complexe instructie onderzoeken:

ADD A, (HL)

De opcode voor de instructie is 10000110. Hij betekent: "tel de inhoud van de accumulator op bij de inhoud van de geheugen locatie (HL)". De geheugen locatie wordt bepaald door een nogal vreemd systeem. Het is de geheugen locatie, waarvan het adres te vinden is in de registers H en L. We nemen dus aan, dat dat adres van te voren in deze registers is geplaatst. De 16 bits grote inhoud van dit register paar is het adres in het geheugen waar de data te vinden is. Deze data wordt bij de accumulator opgeteld, en het resulaat komt weer in de accumulator.

Deze instructie heeft een geschiedenis. De 8008 had deze instructie, en de 8080, de opvolger van de 8008, moest compatibel zijn met zijn

NOTES:

1. The first memory cycle (M1) is always an instruction fetch; the first (or only) byte, containing the op code, is fetched during this cycle.

2. If the READY input from memory is not high during T2 of each memory cycle, the processor will enter a wait state (TW) until READY is sampled as high.

3. States T4 and T5 are present, as required, for operations which are completely internal to the CPU. The contents of the internal bus during T4 and T5 are available at the data bus; this is designed for testing purposes only. An "X" denotes that the state is present, but is only used for such internal operations as instruction decoding.

4. Only register pairs rp = B (registers B and C) or rp = D (registers D and E) may be specified.

5. These states are skipped.

6. Memory read sub-cycles; an instruction or data word will be read.

7. Memory write sub-cycle.

8. The READY signal is not required during the second and third sub-cycles (M2 and M3). The HOLD signal is accepted during M2 and M3. The SYNC signal is not generated during M2 and M3. During the execution of DAD, M2 and M3 are required for an internal register-pair add; memory is not referenced.

9. The results of these arithmetic, logical or rotate instructions are not moved into the accumulator (A) until state T2 of the next instruction cycle. That is, A is loaded while the next instruction is being fetched; this overlapping of operations allows for faster processing.

10. If the value of the least significant 4-bits of the accumulator is greater than 9 or if the auxiliary carry bit is set, 6 is added to the accumulator. If the value of the most significant 4-bits of the accumulator is now greater than 9, or if the carry bit is set, 6 is added to the most significant 4-bits of the accumulator.

11. This represents the first sub-cycle (the instruction fetch) of the next instruction cycle.

12. If the condition was met, the contents of the register pair WZ are output on the address lines (A_{0-15}) instead of the contents of the program counter (PC).

13. If the condition was not met, sub-cycles M4 and M5 are skipped; the processor instead proceeds immediately to the instruction fetch (M1) of the next instruction cycle.

14. If the condition was not met, sub-cycles M2 and M3 are skipped; the processor instead proceeds immediately to the instruction fetch (M1) of the next instruction cycle.

15. Stack read sub-cycle.

16. Stack write sub-cycle.

17. CONDITION

CONDITION	CCC
NZ – not zero (Z = 0)	000
Z – zero (Z = 1)	001
NC – no carry (CY = 0)	010
C – carry (CY = 1)	011
PO – parity odd (P = 0)	100
PE – parity even (P = 1)	101
P – plus (S = 0)	110
M – minus (S = 1)	111

18. I/O sub-cycle: the I/O port's 8-bit select code is duplicated on address lines 0-7 (A_{0-7}) and 8-15 (A_{8-15}).

19. Output sub-cycle.

20. "The processor will remain idle in the halt state until an interrupt, a reset or a hold is accepted. When a hold request is accepted, the CPU enters the hold mode; after the hold mode is terminated, the processor returns to the halt state. After a reset is accepted, the processor begins execution at memory location zero. After an interrupt is accepted, the processor executes the instruction forced onto the data bus (usually a restart instruction).

SSS or DDD	Value
A	111
B	000
C	001
D	010
E	011
H	100
L	101

rp	Value
B	00
D	01
H	10
SP	11

Fig. 2.26: Intel afkortingen

MNEMONIC	OP CODE		M1[1]					M2		
	$D_7 D_6 D_5 D_4$	$D_3 D_2 D_1 D_0$	T1	T2[2]	T3	T4	T5	T1	T2[2]	T3
MOV r1, r2	0 1 D D	D S S S	PC OUT STATUS	PC ← PC + 1	INST → TMP/IR	(SSS) → TMP	(TMP) → DDD			
MOV r, M	0 1 D D	D 1 1 0				X[3]		HL OUT STATUS[6]	DATA	→ DDD
MOV M, r	0 1 1 1	0 S S S				(SSS) → TMP		HL OUT STATUS[7]	(TMP)	→ DATA BUS
SPHL	1 1 1 1	1 0 0 1				(HL)	→ SP			
MVI r, data	0 0 D D	D 1 1 0				X		PC OUT STATUS[6]	B2	→ DDD
MVI M, data	0 0 1 1	0 1 1 0				X			B2	→ TMP
LXI rp, data	0 0 R P	0 0 0 1				X			PC ← PC + 1 B2	→ r1
LDA addr	0 0 1 1	1 0 1 0				X			PC ← PC + 1 B2	→ Z
STA addr	0 0 1 1	0 0 1 0				X			PC ← PC + 1 B2	→ Z
LHLD addr	0 0 1 0	1 0 1 0				X			PC ← PC + 1 B2	→ Z
SHLD addr	0 0 1 0	0 0 1 0				X		PC OUT STATUS[6]	PC ← PC + 1 B2	→ Z
LDAX rp[4]	0 0 R P	1 0 1 0				X		rp OUT STATUS[6]	DATA	→ A
STAX rp[4]	0 0 R P	0 0 1 0				X		rp OUT STATUS[7]	(A)	→ DATA BUS
XCHG	1 1 1 0	1 0 1 1				(HL) ↔ (DE)				
ADD r	1 0 0 0	0 S S S				(SSS) → TMP (A) → ACT		[9]	(ACT) + (TMP) → A	
ADD M	1 0 0 0	0 1 1 0				(A) → ACT		HL OUT STATUS[6]	DATA	→ TMP
ADI data	1 1 0 0	0 1 1 0				(A) → ACT		PC OUT STATUS[6]	PC ← PC + 1 B2	→ TMP
ADC r	1 0 0 0	1 S S S				(SSS) → TMP (A) → ACT		[9]	(ACT) + (TMP) + CY → A	
ADC M	1 0 0 0	1 1 1 0				(A) → ACT		HL OUT STATUS[6]	DATA	→ TMP
ACI data	1 1 0 0	1 1 1 0				(A) → ACT		PC OUT STATUS[6]	PC ← PC + 1 B2	→ TMP
SUB r	1 0 0 1	0 S S S				(SSS) → TMP (A) → ACT		[9]	(ACT) − (TMP) → A	
SUB M	1 0 0 1	0 1 1 0				(A) → ACT		HL OUT STATUS[6]	DATA	→ TMP
SUI data	1 1 0 1	0 1 1 0				(A) → ACT		PC OUT STATUS[6]	PC ← PC + 1 B2	→ TMP
SBB r	1 0 0 1	1 S S S				(SSS) → TMP (A) → ACT		[9]	(ACT) − (TMP) − CY → A	
SBB M	1 0 0 1	1 1 1 0				(A) → ACT		HL OUT STATUS[6]	DATA	→ TMP
SBI data	1 1 0 1	1 1 1 0				(A) → ACT		PC OUT STATUS[6]	PC ← PC + 1 B2	→ TMP
INR r	0 0 D D	D 1 0 0				(DDD) → TMP (TMP) + 1 → ALU	ALU → DDD			
INR M	0 0 1 1	0 1 0 0				X		HL OUT STATUS[6]	DATA (TMP) + 1	→ TMP → ALU
DCR r	0 0 D D	D 1 0 1				(DDD) → TMP (TMP) + 1 → ALU	ALU → DDD			
DCR M	0 0 1 1	0 1 0 1				X		HL OUT STATUS[6]	DATA (TMP) − 1	→ TMP → ALU
INX rp	0 0 R P	0 0 1 1				(RP) + 1	→ RP			
DCX rp	0 0 R P	1 0 1 1				(RP) − 1	→ RP			
DAD rp[8]	0 0 R P	1 0 0 1				X		(ri) → ACT	(L) → TMP, (ACT) + (TMP) → ALU	ALU → L, CY
DAA	0 0 1 0	0 1 1 1				DAA → A, FLAGS[10]				
ANA r	1 0 1 0	0 S S S				(SSS) → TMP (A) → ACT		[9]	(ACT) + (TMP) → A	
ANA M	1 0 1 0	0 1 1 0	PC OUT STATUS	PC ← PC + 1	INST → TMP/IR	(A) → ACT		HL OUT STATUS[6]	DATA	→ TMP

Fig. 2.27: Intel instructie formaten

M3			M4			M5				
T1	T2[2]	T3	T1	T2[2]	T3	T1	T2[2]	T3	T4	T5
HL OUT STATUS[7]	(TMP)	→ DATA BUS								
PC OUT STATUS[6]	PC = PC + 1 B3	→ rh								
↑	PC = PC + 1 B3	→ W	WZ OUT STATUS[6]	DATA	→ A					
│	PC = PC + 1 B3	→ W	WZ OUT STATUS[7]	(A)	→ DATA BUS					
↓	PC = PC + 1 B3	→ W	WZ OUT STATUS[6]	DATA WZ = WZ + 1	→ L	WZ OUT STATUS[6]	DATA	→ H		
PC OUT STATUS[6]	PC = PC + 1 B3	→ W	WZ OUT STATUS[7]	(L) WZ = WZ + 1	→ DATA BUS	WZ OUT STATUS[7]	(H)	→ DATA BUS		
(9)	(ACT)+(TMP)→A									
(9)	(ACT)+(TMP)→A									
(9)	(ACT)+(TMP)+CY→A									
(9)	(ACT)+(TMP)+CY→A									
(9)	(ACT)-(TMP)→A									
(9)	(ACT)-(TMP)→A									
(9)	(ACT)-(TMP)-CY→A									
(9)	(ACT)-(TMP)-CY→A									
HL OUT STATUS[7]	ALU	→ DATA BUS								
HL OUT STATUS[7]	ALU	→ DATA BUS								
(rh)→ACT	(H)→TMP (ACT)+(TMP)+CY→ALU	ALU→H, CY								
(9)	(ACT)+(TMP)→A									

Fig. 2.27: Intel instructie formaten (vervolg)

MNEMONIC	OP CODE		M1[1]					M2		
	$D_7D_6D_5D_4$	$D_3D_2D_1D_0$	T1	T2[2]	T3	T4	T5	T1	T2[2]	T3
ANI data	1 1 1 0	0 1 1 0	PC OUT STATUS	PC = PC + 1	INST→TMP/IR	(A)→ACT		PC OUT STATUS[6]	PC = PC + 1 B2	→TMP
XRA r	1 0 1 0	1 S S S				(A)→ACT (SSS)→TMP		[9]	(ACT)+(TMP)→A	
XRA M	1 0 1 0	1 1 1 0				(A)→ACT		HL OUT STATUS[6]	DATA	→TMP
XRI data	1 1 1 0	1 1 1 0				(A)→ACT		PC OUT STATUS[6]	PC = PC + 1 B2	→TMP
ORA r	1 0 1 1	0 S S S				(A)→ACT (SSS)→TMP		[9]	(ACT)+(TMP)→A	
ORA M	1 0 1 1	0 1 1 0				(A)→ACT		HL OUT STATUS[6]	DATA	→TMP
ORI data	1 1 1 1	0 1 1 0				(A)→ACT		PC OUT STATUS[6]	PC = PC + 1 B2	→TMP
CMP r	1 0 1 1	1 S S S				(A)→ACT (SSS)→TMP	x	[9]	(ACT)-(TMP), FLAGS	
CMP M	1 0 1 1	1 1 1 0				(A)→ACT		HL OUT STATUS[6]	DATA	→TMP
CPI data	1 1 1 1	1 1 1 0				(A)→ACT		PC OUT STATUS[6]	PC = PC + 1 B2	→TMP
RLC	0 0 0 0	0 1 1 1				(A)→ALU ROTATE		[9]	ALU→A, CY	
RRC	0 0 0 0	1 1 1 1				(A)→ALU ROTATE		[9]	ALU→A, CY	
RAL	0 0 0 1	0 1 1 1				(A), CY→ALU ROTATE		[9]	ALU→A, CY	
RAR	0 0 0 1	1 1 1 1				(A), CY→ALU ROTATE		[9]	ALU→A, CY	
CMA	0 0 1 0	1 1 1 1				(A)→A				
CMC	0 0 1 1	1 1 1 1				$\overline{CY}$→CY				
STC	0 0 1 1	0 1 1 1				1→CY				
JMP addr	1 1 0 0	0 0 1 1				x		PC OUT STATUS[6]	PC = PC + 1 B2	→Z
J cond addr[17]	1 1 C C	C 0 1 0				JUDGE CONDITION		PC OUT STATUS[6]	PC = PC + 1 B2	→Z
CALL addr	1 1 0 0	1 1 0 1				SP = SP - 1		PC OUT STATUS[6]	PC = PC + 1 B2	→Z
C cond addr[17]	1 1 C C	C 1 0 0				JUDGE CONDITION IF TRUE, SP = SP - 1		PC OUT STATUS[6]	PC = PC + 1 B2	→Z
RET	1 1 0 0	1 0 0 1				x		SP OUT STATUS[15]	SP = SP + 1 DATA	→Z
R cond addr[17]	1 1 C C	C 0 0 0			INST→TMP/IR	JUDGE CONDITION[14]		SP OUT STATUS[15]	SP = SP + 1 DATA	→Z
RST n	1 1 N N	N 1 1 1			0→W INST→TMP/IR	SP = SP - 1		SP OUT STATUS[16]	SP = SP - 1 (PCH)	→DATA BUS
PCHL	1 1 1 0	1 0 0 1			INST→TMP/IR	(HL)	→PC			
PUSH rp	1 1 R P	0 1 0 1				SP = SP - 1		SP OUT STATUS[16]	SP = SP - 1 (rh)	→DATA BUS
PUSH PSW	1 1 1 1	0 1 0 1				SP = SP - 1		SP OUT STATUS[16]	SP = SP - 1 (A)	→DATA BUS
POP rp	1 1 R P	0 0 0 1				x		SP OUT STATUS[15]	SP = SP + 1 DATA	→rl
POP PSW	1 1 1 1	0 0 0 1				x		SP OUT STATUS[15]	SP = SP + 1 DATA	→FLAGS
XTHL	1 1 1 0	0 0 1 1				x		SP OUT STATUS[15]	SP = SP + 1 DATA	→Z
IN port	1 1 0 1	1 0 1 1				x		PC OUT STATUS[6]	PC = PC + 1 B2	→Z, W
OUT port	1 1 0 1	0 0 1 1				x		PC OUT STATUS[6]	PC = PC + 1 B2	→Z, W
EI	1 1 1 1	1 0 1 1				SET INTE F/F				
DI	1 1 1 1	0 0 1 1				RESET INTE F/F				
HLT	0 1 1 1	0 1 1 0				x		PC OUT STATUS	HALT MODE[20]	
NOP	0 0 0 0	0 0 0 0	PC OUT STATUS	PC = PC + 1	INST→TMP/IR	x				

Fig. 2.27: Intel instructie formaten (vervolg)

M3			M4			M5						
T1	T2[2]	T3	T1	T2[2]	T3	T1	T2[2]	T3	T4	T5		
[9]	(ACT)+(TMP)→A											
[9]	(ACT)+(TMP)→A											
[9]	(ACT)+(TMP)→A											
[9]	(ACT)+(TMP)→A											
[9]	(ACT)+(TMP)→A											
[9]	(ACT)-(TMP), FLAGS											
[9]	(ACT)-(TMP), FLAGS											
PC OUT STATUS[6]	PC = PC + 1 B3 →	W									WZ OUT STATUS[11]	(WZ) + 1 → PC
PC OUT STATUS[6]	PC = PC + 1 B3 →	W									WZ OUT STATUS[11,12]	(WZ) + 1 → PC
PC OUT STATUS[6]	PC = PC + 1 B3 →	W	[illegible] OUT STATUS[16]	(PCH) → SP = SP - 1	DATA BUS	SP OUT STATUS[16]	(PCL) →	DATA BUS			WZ OUT STATUS[11]	(WZ) + 1 → PC
PC OUT STATUS[6]	PC = PC + 1 B3 →	W[13]	SP OUT STATUS[16]	(PCH) SP = SP - 1	→ DATA BUS	SP OUT STATUS[16]	(PCL) →	DATA BUS			WZ OUT STATUS[11,12]	(WZ) + 1 → PC
SP OUT STATUS[15]	SP = SP + 1 DATA →	W									WZ OUT STATUS[11]	(WZ) + 1 → PC
SP OUT STATUS[15]	SP = SP + 1 DATA →	W									WZ OUT STATUS[11,12]	(WZ) + 1 → PC
SP OUT STATUS[16]	(TMP = 00NNN000) → (PCL) →	Z LATA BUS									WZ OUT STATUS[11]	(WZ) + 1 → PC
SP OUT STATUS[16]	(H) →	DATA BUS										
SP OUT STATUS[16]	FLAGS →	DATA BUS										
SP OUT STATUS[15]	SP = SP + 1 DATA →	h										
SP OUT STATUS[15]	SP = SP + 1 DATA →	A										
SP OUT STATUS[15]	DATA →	W	SP OUT STATUS[16]	(H) →	DATA BUS	SP OUT STATUS[16]	(L) →	DATA BUS	(WZ) →	HL		
WZ OUT STATUS[16]	DATA →	A										
WZ OUT STATUS[16]	(A) →	DATA BUS										

Fig. 2.27: Intel instructie formaten (vervolg)

voorganger. De 8008 had geen directe geheugen adressering! Om het geheugen te adresseren moesten eerst de registers H en L geladen worden met het adres. ADD A,(HL) is zo'n instructie waar dat bij nodig was (en is). De 8080 en de Z80 hebben meer adresserings mogelijkheden en bij deze microprocessors is deze manier van adresseren beslist geen beperking, zoals die het was bij de 8008, doch veeleer een voordeel.

Laten we de executie van de instructie eens bekijken (deze instructie heet bij de 8080 ADD M en is de 16e instructie van figuur 2.27). T1, T2 en T3 van M1 worden gebruikt voor de fetch van de instructie. Tijdens T4 komt de inhoud van de accumulator in een buffer, ACT, en aldus op de linker ingang van de ALU.

Het tweede getal voor de optelling staat in het geheugen. H en L bevatten het adres. De inhoud van deze registers moet dus op de adres bus worden gezet.

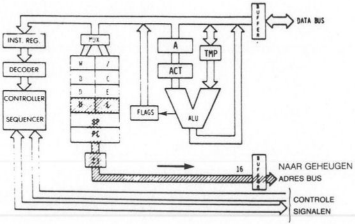


Fig. 2.28: De inhoud van HL komt op de adres bus

In de tabel lezen we: HL OUT (tijdens M2). H en L worden op de zelfde wijze als PC bij andere instructies op de adres bus gezet. Er is al opgemerkt, dat gedurende T1 *status* op de data bus staat. Hier maken we er geen gebruik van. Vereenvoudigd kunnen we stellen, dat er twee states nodig zijn: een om de data te lezen uit het geheugen, en een om de data te transporteren naar de rechter ingang van de ALU, TMP.

Beide getallen zijn nu aanwezig op de ingangen van de ALU. De situatie is nu gelijk aan die bij ADD A,r. Een fetch-executie overlapping wordt gebruikt, en in plaats van de instructie af te maken in T4 van M2, gebeurt dit tijdens T2 van M3. Dit alles is terug te vinden in figuur 2.27.

Vraag: Wat is de schijnbare tijdsduur (voor de programmeur) van deze instructie? Is dat 7,5 of 4,5 microseconden?

We zullen nog een complexe instructie bekijken met directe geheugen adressering en twee onzichtbare registers W en Z:

LD A,(nn)

De opcode is 00111010. Het 8080 equivalent is LDA adr. T1, T2, T3 van M1 worden zoals gebruikelijk benut voor de instructie fetch. T4 wordt ook gebruikt, maar zonder zichtbaar resultaat. In feite wordt dan de instructie gedecodeerd. De contole eenheid komt er dan achter dat nog twee bytes uit het geheugen moeten worden gehaald. Deze twee bytes bevatten een adres. De accumulator moet geladen worden met de inhoud van het geheugen op dat adres. Merk op, dat T4 nodig is om de instructie te *decoderen*. Je zou dat tijdverspilling kunnen noemen, omdat slechts een deel van T4 gebruikt wordt voor het decoderen. Dit is echter de filosofie van *klok-synchrone logica*. Omdat *microinstructies* gebruikt worden om intern de instructie te decoderen en te executeren is dat de prijs, die we moeten betalen, in ruil voor alle voordelen die microprogrammering ons biedt. De structuur van deze instructie staat in figuur 2.29.

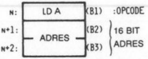


Fig. 2.29: LD A, (ADRES) is een drie-woords instructie

De volgende twee bytes worden opgehaald. Ze vormen tesamen een adres (Zie figuur 2.30).

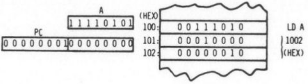


Fig. 2.30: Voor executie van LD A

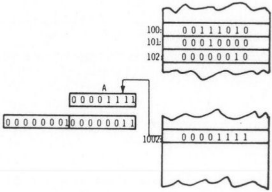


Fig. 2.31: Na executie van LD A

Het effect van de instructie is te zien in de figuren 2.30 en 2.31.
De controle eenheid heeft de beschikking over twee registers in de Z80 (de programmeur heeft dat niet). Het zijn de registers W en Z, te vinden in figuur 2.28.

Tweede machine cyclus M2: T1 en T2 worden op de gebruikelijke manier gebruikt om de inhoud van geheugen locatie PC te halen. Tijdens T2 wordt bij PC 1 opgeteld. Ergens aan het einde van T2 komt de data van het geheugen beschikbaar, en verschijnt op de data bus. Aan het eind van T3 wordt dit woord (B2:tweede woord van de instructie) opgeborgen in een tijdelijk register. Het komt in Z terecht: B2 ► Z. (Zie figuur 2.32).

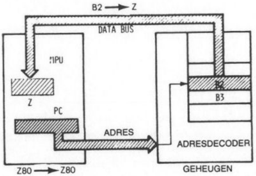


Fig. 2.32: Tweede woord van de instructie komt in Z

Machine cyclus M3: Op dezelfde wijze als B2 in Z terecht komt, komt B3 in W. Aan het eind van T3 van M3 bevatten W en Z het adres, dat oorspronkelijk in de laatste twee woorden van de instructie stond. We kunnen de instructie nu afmaken. W en Z bevatten een adres. Dit adres moet naar het geheugen worden gestuurd, om de data op te kunnen halen. Dit wordt in de volgende geheugen cyclus gedaan:

Machine cyclus M4: W en Z worden op de adres bus gezet. Aan het eind van T2 is de data op de data bus beschikbaar. Tijdens T3 wordt deze data in de accumulator opgeslagen. Daarmee is deze instructie beeindigd.

Dit was een voorbeeld van een *onmiddellijke instructie*. De instructie is drie bytes lang om een twee-bytes expliciet adres te kunnen bevatten. De instructie gebruikt vier geheugen cycli. Het is een lange instructie. Maar we kunnen stellen, dat het een fundamentele instructie is voor

het laden van de accumulator met een gespecificeerde inhoud van een bekende geheugen locatie. De gebruikte registers zijn W en Z.

Vraag: Zou de instructie andere registers dan W en Z hebben kunnen gebruiken?

Antwoord: Nee. Als de instructie andere registers zou gebruiken, bijvoorbeeld H en L, dan zou hun inhoud veranderen. En de aanname bij instructies is, dat inhouden van registers die niet expliciet worden gebruikt, niet mogen veranderen. Een instructie, die de accumulator laadt, mag de inhoud van de andere registers niet vernietigen. Daarom heeft het systeem de beschikking over deze extra registers voor intern gebruik.

Vraag: Is het mogelijk PC te gebruiken in plaats van W en Z?

Antwoord: Zeer zeker niet. Dat zou zelfmoord betekenen. De lezer moet dit maar voor zichzelf nagaan.

We zullen nog een instructie meer bekijken: de *sprong* instructie (*branch* of *jump*), waardoor de volgorde van de instructies in een programma wordt veranderd. Tot nu toe hebben we aangenomen, dat de instructies sequentieel worden doorlopen. Er bestaan echter instructies, waardoor we naar andere instructies kunnen springen in een ander deel van het geheugen. Zo'n instructie is:

JP nn

Deze instructie staat op regel 18 van figuur 2.27 als "JMP−addr". De uitvoering ervan wordt beschreven door de tabel horizontaal te volgen. Ook dit is een drie-woords instructie. Het eerste woord is de opcode: 11000011. De volgende twee woorden bevatten het adres waar naar toe gesprongen wordt. Het principe is, dat de inhoud van de programma teller vervangen wordt door de 16 bits, die volgen op de opcode. Om de instructie efficient uit te voeren, wordt in de praktijk vaak een iets andere benadering gevolgd. Zoals altijd worden de eerste drie states van M1 gebruikt voor de fetch. Tijdens T4 wordt de instructie gedecodeerd en wordt er geen andere gebeurtenis waargenomen (X). De volgende machine cycli worden gebruikt om B2 en B3 van de instructie op te halen. Tijdens M2 wordt B2 gehaald en in W gezet. De volgende twee stappen vinden plaats tijdens de fetch van de volgende instructie,

net zoals dat gebeurde bij de optelling. Zij worden daarna uitgevoerd in plaats van de gebruikelijke T1 en T2 van de volgende instructie. Laten we ze eens bekijken.

De volgende stappen zijn: WZ OUT en (WZ) + 1 ▶ PC. Met andere woorden, de inhoud van WZ wordt gebruikt in plaats van die van PC tijdens de volgende instructie fetch. De controle eenheid heeft waargenomen, dat er een sprong plaats vindt, en zal de volgende instructie anders beginnen.

Het effect van de twee extra states is:

Het adres op de adres bus is afkomstig van WZ. De volgende instructie is dus afkomstig van het adres, dat in WZ zat. Het resultaat is een sprong. Daarna wordt bij de inhoud van WZ 1 opgeteld en in de programma teller geplaatst. De volgende instructie wordt dus normaal door PC bepaald.

Vraag: Waarom laden we niet direct de inhoud van PC? Waarom moeten we W en Z gebruiken?

Antwoord: Het is onmogelijk PC te gebruiken. Als het lage orde deel van PC (PTL) geladen wordt met B2, dan vernietigen we de oorspronkelijke inhoud van PC. Het is daarna onmogelijk B3 te halen uit het geheugen.

Vraag: Is het mogelijk alleen Z te gebruiken, in plaats van W en Z?

Antwoord: Ja, maar het zou langzamer gaan. We zouden Z kunnen laden met B2, om daarna het hoge orde deel van PC (PCH) met B3 te laden. Dan is het nog nodig PCL te laden met Z. Dit alles zou het gehele proces erg vertragen. Daarom moeten W en Z worden gebruikt. Bovendien worden, om tijd te sparen, W en Z niet naar PC gebracht, maar worden direct op de adres bus gezet om de nieuwe instructie te halen. Dit punt is erg belangrijk om het efficient uitvoeren van instructies binnen de microprocessor te begrijpen.

Vraag: (Alleen voor lezers met ervaring op microprocessor gebied). Wat gebeurt er als er een interrupt plaats vindt aan het eind van M3? (Als de executie van het programma wordt onderbroken op dit punt, dan wijst de PC naar de instructie volgend op de sprong, en het spronggadres in W en Z zal verloren gaan).

Het antwoord wordt als een interessante oefening aan de desbetreffende lezers overgelaten.

De rol van de interne registers en bussen moet uit deze voorbeelden duidelijk zijn geworden. Om de interne werking van de Z80 te begrijpen moet dit stuk misschien nog eens worden doorgenomen.

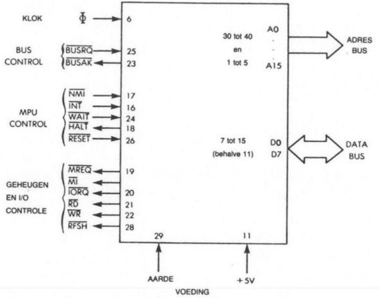


Fig. 2.33: Z80 MPU pinuitvoering

De Z80 chip

Voor alle volledigheid zullen we de signalen van de Z80 microprocessor chip hier bekijken. Om de Z80 te kunnen programmeren hoeven we deze signalen eigenlijk niet te kennen. De niet geinteresseerde lezer kan dit deel dus overslaan. De pinuitvoering staat in figuur 2.33. Rechts in de figuur zijn de data en adres bus te vinden. Hier bespreken we de signalen van de controle bus. Ze staan links in de figuur.

De controle signalen zijn verdeeld in vier groepen. We zullen ze behandelen van boven naar beneden.

De klok ingang is 0. De klokoscillator zit op de Z80 chip. Extern is alleen een 330 ohm pull-up weerstand nodig. Deze moet verbonden worden met ingang 0 en 5 volt. Voor 4 MHz is echter een externe klok driver nodig.

De twee bus-controle signalen, BUSRQ en BUSAK, worden gebruikt om de Z80 van de bussen af te schakelen. Hoofdzakelijk bedoeld voor DMA, maar ze kunnen ook gebruikt worden door een tweede processor in het systeem. BUSRQ is het signaal om de bus aan te vragen. Dit signaal gaat naar de Z80. Als reactie op dit signaal zet de Z80 de adres bus, de data bus en de tristate controle uitgangen in de hoge impedantie toestand, aan het eind van de huidige machine cyclus. Als dat is gebeurd, zendt de Z80 een bevestiging: BUSAK

Zes controle lijnen houden verband met de interne status van de Z80:

INT en NMI zijn twee interrupt signalen. INT is de normale interrupt aanvraag. Interrupts komen in hoofdstuk 6 aan de orde. Een aantal input/output apparaten mogen met INT worden verbonden. Als een interrupt aanvraag op deze lijn aanwezig is, en interrupts zijn toegestaan, dan zal de Z80 deze accepteren (aangenomen dat BUSRQ niet actief is). De Z80 genereert een bevestiging, IORQ, tijdens M1. De rest van de gebeurtenissen is beschreven in hoofdstuk 6.

NMI is de niet-maskeerbare interrupt. Deze wordt altijd door de Z80 geaccepteerd, en dwingt de processor naar 0066 hexadecimaal te springen. (Ook hier geldt, dat BUSRQ niet actief mag zijn.)

WAIT is een signaal, dat gebruikt wordt om de Z80 met langzame geheugens of I/O apparaten te synchroniseren. Als dit signaal actief is, betekent het, dat het geheugen of apparaat nog niet klaar is voor data overdracht. De Z80 komt terecht in een speciale toestand, de "wait state", totdat WAIT niet meer actief is. De normale gang van zaken wordt dan weer hervat.

HALT is een bevestiging, nadat de Z80 een HALT instructie heeft geexecuteerd. In deze toestand wacht de processor op een externe interrupt en blijft NOP's uitvoeren voor de refresh van het geheugen.

RESET is het signaal waarmee gewoonlijk de MPU wordt geinitialiseerd. Het zet de programma teller, register I en R op "0". Het reset de "interrupt enable flip-flop" en zet de interrupt mode op "0". Meestal wordt het gebruikt als de voeding is aangezet.

Geheugen en I/O controle

Er worden zes geheugen en I/O controle signalen gemaakt:

MREQ is het geheugen aanvraag signaal. Het geeft aan, dat het adres op de adres bus geldig is. Een lees of schrijf operatie kan worden uitgevoerd.

M1 is machine cyclus 1. Het correspondeert met de fetch cyclus van een instructie.

IORQ is een I/O aanvraag. Het geeft aan, dat het I/O adres op de bits 0-7 van de adres bus geldig zijn. Een I/O lees of schrijf operatie kan worden uitgevoerd. IORQ wordt ook tesamen met M1 gegenereerd als de Z80 een interrupt bevestigt. Deze informatie kan door externe chips gebruikt worden om de interrupt vector op de data bus te plaatsen. (De combinatie M1 en IORQ duidt op een bevestiging van een interrupt, omdat normale I/O operaties niet voorkomen tijdens M1.)

RD is het geheugen lees signaal. De Z80 is dan gereed om data van de data bus in de accumulator te plaatsen. Het kan door iedere externe chip gebruikt worden, geheugen of I/O, om data op de data bus te plaatsen.

WR is het geheugen schrijf signaal. De data op de data bus is geldig. Het gespecificeerde apparaat mag de data accepteren.

RFSH is het refresh signaal. Als dit signaal actief is, dan bevatten de laagste zeven adres bits het refresh adres voor dynamische geheugens. Het MREQ signaal voert de refresh uit, door het geheugen dan te lezen

SAMENVATTING VAN DE HARDWARE

We zijn nu klaar met de beschrijving van de interne organisatie van de Z80. De exacte details van de Z80 hardware zijn hier niet belangrijk. De rol van de registers is dat echter wel en moet volledig begrepen zijn voordat begonnen kan worden met de volgende hoofdstukken. Straks komen alle instructies van de Z80 aan de orde, maar eerst houden we ons in het volgende hoofdstuk bezig met de basis programmeer technieken.

3
BASIS PROGRAMMEER TECHNIEKEN

INLEIDING

Doel van dit hoofdstuk is de noodzakelijke basis technieken te geven om een Z80 programma te kunnen schrijven. In dit hoofdstuk zullen enkele nieuwe grondbeginselen worden geintroduceerd, zoals register beheer, loops en subroutines. We zullen vooral de aandacht richten op die technieken, die alleen gebruik maken van de *interne* registers. Echte programma's worden hier ontwikkeld, zoals rekenkundige programma's. Deze programma's worden gebruikt als illustratie van de tot dusver gepresenteerde grondbeginselen. Op die manier wordt gedemonstreerd hoe de instructies worden gebruikt om de data tussen geheugen en MPU, en binnen de MPU zelf te manipuleren. In het volgende hoofdstuk worden dan alle instructies van de Z80 tot in details besproken. Hoofdstuk 5 behandelt de adresserings technieken, en hoofdstuk 6 gaat over de manier waarop data *buiten* de Z80 gemanipuleerd kunnen worden: de input/output technieken.

In dit hoofdstuk geldt het devies: al doende leert men. Door steeds moeilijker programma's te onderzoeken zullen we de rol van de instructies en registers leren kennen, en we zullen de tot dusver ontwikkelde grondbeginselen toepassen. Een belangrijk grondbeginsel wordt hier niet behandeld: de adresserings technieken. Dat gebeurt in hoofdstuk 5.

We beginnen meteen met het schrijven van programma's. Als eerste rekenkundige. Het "programmeurs model" van de Z80 registers wordt in figuur 3.0 getoond.

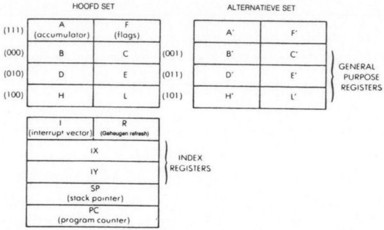


Fig. 3.0: De Z80 registers

REKENKUNDIGE PROGRAMMA'S

Tot de rekenkundige programma's behoren optellen, aftrekken, vermenigvuldigen en delen. De hier gepresenteerde programma's werken met gehele getallen. Deze gehele getallen mogen positief binair zijn, maar mogen ook worden gepresenteerd in het twee-complement. In dat geval is het meest linkse bit het tekenbit. (Zie hoofdstuk 1 voor een beschrijving van deze notatie).

8-bits optelling

We zullen twee 8-bits grote getallen, OP1 en OP2, optellen. Deze getallen zijn opgeslagen op de geheugen adressen ADR1 en ADR2. De som heet RES en komt op adres ADR3. Dit wordt geillustreerd in figuur 3.1. Het programma, dat de optelling uitvoerd is het volgende:

```
Instructies         Commentaar

LD    A,(ADR1)      ;LAAD OP1 IN A
LD    HL,ADR2       ;LAAD ADRES VAN OP2 IN HL
ADD   A,(HL)        ;TEL OP2 BIJ OP1 OP
LD    (ADR3),A      ;ZET HET RESULTAAT OP ADR3
```

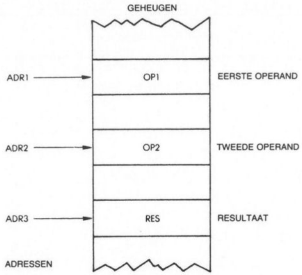


Fig. 3.1: 8-bits optelling RES = OP1 + OP2

Dit is ons eerste programma. De instructies staan links en het commentaar rechts. Laten we het programma eens bekijken. Het is een programma met vier instructies. Iedere regel wordt *instructie* genoemd en is weergegeven in een symbolische vorm. Iedere instructie wordt vertaald door een *assembler* programma in een, twee, drie of vier bytes. We houden ons hier niet bezig met de vertaling en zullen alleen naar de symbolische voorstelling kijken.

In de eerste regel wordt de inhoud vañ ADR1 in de accumulator geladen. In figuur 3.1 is te zien, dat dat OP1 is. Het resultaat van deze eerste instructie is dus, dat OP1 van het geheugen naar de accumulator wordt verplaatst. Dit is in figuur 3.2 weergegeven. "ADR1" is een symbolische voorstelling van een werkelijk 16-bits adres in het geheugen. Ergens anders in een programma zal dat adres worden gedefinieerd. ADR1 zou, bijvoorbeeld, gelijk kunnen worden gemaakt aan adres "100".

Een *lees* cyclus is het resultaat van deze *load* (laad) instructie. De inhoud van adres 100 wordt gelezen en getransporteerd via de data bus naar de accumulator, waar het wordt geladen (Zie figuur 3.2). Van het

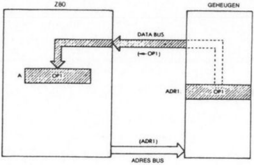


Fig. 3.2: LD A, (ADR1): OP1 is geladen uit het geheugen

vorige hoofdstuk weten we nog, dat een van de operanden van een rekenkundige of logische bewerking altijd de accumulator is. (Lees het vorige hoofdstuk nog maar eens door voor meer details). Omdat we twee waarden op willen tellen, moeten we eerst OP1 in de accumulator laden. Daarna kunnen we OP2 erbij optellen. Het meest rechtse veld van de instructie is het *commentaar* veld. Bij de vertaling door het assembler programma wordt dit deel van de instructie genegeerd, maar is aan de instructie toegevoegd, om het leesbaar te maken. Om goed te kunnen begrijpen wat een programma doet, is het erg belangrijk, dat er goed commentaar bij staat. Het toevoegen van commentaar heet *documenteren*.

Hier legt het commentaar uit wat er gedaan wordt: de waarde van OP1 op adres ADR1 wordt in de accumulator geladen.

Het resultaat van deze instructie zien we in figuur 3.2. De tweede instructie van ons programma luidt:

```
LD HL, ADR2
```

Dit betekent: laadt ADR2 in de registers H en L. Om de tweede operand, OP2, uit het geheugen te kunnen lezen, moeten we eerst het adres ervan in een registerpaar van de Z80 plaatsen, bijvoorbeeld H en L. Daarna kunnen we de inhoud van dit geheugen adres bij de accumulator optellen.

In figuur 3.1 zien we, dat de inhoud van het geheugen adres ADR2 OP2 is, onze tweede operand. De inhoud van de accumulator is OP1, de eerste operand. Door de instructie wordt OP2 uit het geheugen gehaald en bij OP1 opgeteld. Dit is te zien in figuur 3.3.

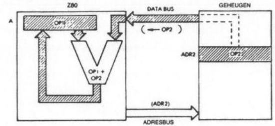


Fig. 3.3: ADD A, (HL)

De som wordt in de accumulator geplaatst. De lezer zal zich ongetwijfeld herinneren, dat het resultaat van een rekenkundige bewerking altijd in de accumulator terecht komt. Bij andere processors is het soms mogelijk het resultaat in een ander register of het geheugen te plaatsen.

De som van OP1 en OP2 bevindt zich in de accumulator. Om ons programma af te maken, moeten we de inhoud van de accumulator naar geheugen adres ADR3 brengen. Dat doet de vierde instructie:

LD (ADR3), A

Deze instructie laadt het gespecificeerde adres ADR3 met de inhoud van de accumulator. Zie figuur 3.4.

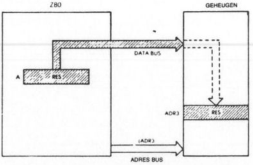


Fig. 3.4: LD (ADR3), A

Voor de uitvoering van de ADD instructie bevatte de accumulator OP1 (Zie figuur 3.3). Na de optelling "OP1 + OP2". Men zij er nogmaals aan herinnerd, dat ieder register of iedere geheugenplaats hetzelfde blijft na een lees operatie. Of met andere woorden, het lezen van een register of het geheugen verandert de inhoud ervan niet. Uitsluitend en alleen een schrijf operatie verandert de inhoud. In dit voorbeeld blijven de inhouden van ADR1 en ADR2 onveranderd. Omdat echter het resultaat van de optelling in A terecht komt, gaat de oorspronkelijke inhoud verloren.

Echte adressen mogen gebruikt worden in plaats van ADR1, ADR2 en ADR3. Om deze symbolische adressen te kunnen gebruiken zijn "pseudo-instructies" nodig, die de waarde van deze adressen specificeren. Tijdens het vertalen worden de symbolische adressen vervangen door de echte waarden. Zulke pseudo-instructies zouden bijvoorbeeld kunnen zijn:

ADR1 = 100H
ADR2 = 120H
ADR3 = 200H

Oefening 3.1: Doe nu het boek dicht. Raadpleeg alleen als dat nodig is de lijst met instructies achter in het boek. Schrijf een programma dat twee getallen optelt, die opgeslagen zijn op de geheugen adressen LOC1 en LOC2. Plaats het resultaat op adres LOC3. Vergelijk daarna dit programma met het net behandelde programma.

16-bits optelling

Bij een 8-bits optelling kunnen we alleen 8-bits getallen optellen, getallen dus tussen 0 en 255 als de binaire voorstelling wordt gebruikt In de meeste toepassingen moeten echter 16-bits of grotere getallen worden opgeteld. We moeten dus meervoudige precisie gebruiken. Er zullen enkele voorbeelden gegeven worden van het rekenen met 16-bits getallen. Dit kan eenvoudig uitgebreid worden tot 24-, 32-bits en nog grotere getallen (altijd veelvouden van 8 bits). We nemen aan, dat de eerste operand op de geheugen adressen ADR1 en ADR1-1 staat. Omdat het een 16-bits getal is zijn twee geheugen adressen nodig. Op de zelfde wijze bevindt OP2 zich op ADR2 en ADR2-1. Het resultaat komt in ADR3 en ADR3-1. Zie figuur 3.5. H betekent het hoge orde deel en L het lage orde deel van het getal.

De eerste vier instructies zijn gelijk aan die van de 8-bits optelling in de vorige sectie. Hier worden de minst significante delen (bits 0-7) van OP1 en OP2 opgeteld. De som, "RES" komt op geheugenadres ADR3 te staan. Zie figuur 3.5.

Automatisch wordt bij iedere optelling het carry bit gered. Dit kan een 0 of een 1 zijn. Dat gebeurt in bit C van het status register. Wordt een carry gegenereerd, dan krijgt C de waarde 1, anders de waarde 0.

De volgende vier instructies zijn in wezen gelijk aan de vorige vier. Nu worden de meest significante helften van de getallen (bits 8-15) plus het carrybit opgeteld en het resultaat komt op ADR3-1 te staan.

Als dit 8 bits lange programma is uitgevoerd, is het resultaat in het geheugen te vinden op de adressen ADR3 en ADR3-1. Merk echter op, dat de beide helften van het programma niet geheel gelijk zijn. Er is een verschil. De "ADD" instructie in de eerste helft is in de tweede helft vervangen door "ADC". ADD telt twee getallen op, ongeacht het carrybit. ADC telt twee getallen op plus het carrybit.

De vraag die nu rijst is: wat gebeurt er als de tweede optelling ook een carry genereert? Er zijn twee mogelijkheden: ten eerste, we nemen aan dat dit een fout is. Er moet dus voor gezorgd worden dat het resultaat altijd 16 bits groot of kleiner is. De tweede mogelijkheid is een test in te bouwen op een eventuele carry aan het eind van het programma. Dit is een keuze die de programmeur moet maken, de eerste van de vele die volgen.

NB: we hebben aangenomen, dat het meest significante deel opgeslagen is "boven" het minst significante deel, d.w.z. op een lager adres in het geheugen. In feite gebeurt in de Z80 precies het omgekeerde bij het opslaan van adressen: eerst wordt het lagere orde deel gered, daarna het hogere orde deel op het volgende adres. Om data en adressen gelijk te behandelen wordt het aanbevolen het minst significante deel van de data op het meest significante deel te plaatsen. Dit is te zien in figuur 3.6.

Als we werken met meer bytes operanden moeten we twee afspraken goed in gedachten houden:

- de volgorde waarin data in het geheugen is geplaatst,
- waar de datapointers naar wijzen: het hoge- of lage orde deel.

Oefeningen 3.2 en 3.3 lichten dit nader toe.

Oefening 3.2: Herschrijf het programma zo, dat figuur 3.6 van toepassing is.

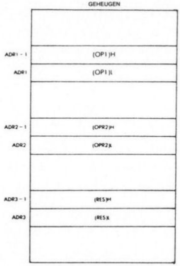


Fig. 3.5: 16-bits optelling. De operanden

De logica van het programma is gelijk aan het vorige programma. Allereerst worden de lagere orde delen van de operanden opgeteld, omdat de microprocessor slechts 8 bits tegelijk kan optellen. Een door de optelling gegenereerde carry wordt automatisch opgeslagen in het interne carrybit ("C"). Daarna worden de hogere orde delen van de operanden opgeteld samen met een eventuele carry. Het resultaat komt in het geheugen. Het programma volgt nu:

```
LD   A, (ADR1)      LAAD LAGERE ORDE DEEL VAN OP1
LD   HL, ADR2       ADRES VAN LAGERE ORDE DEEL
                    VAN OP2
ADD A, (HL),        TEL LAGERE ORDE DELEN VAN OP1
                    EN OP2 OP
LD   (ADR3), A      ZET HET RESULTAAT WEG
LD   A, (ADR1-1)    LAAD HOGERE ORDE DEEL VAN OP1
DEC HL              ADRES HOGERE ORDE DEEL VAN OP2
ADC A, (HL)         (OP1 + OP2)HOOG PLUS CARRY
LD   (ADR3-1), A    ZET HOGERE ORDE DEEL RESULTAAT
                    WEG
```

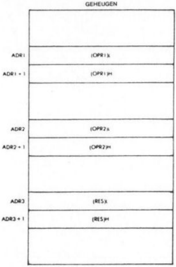


Fig. 3.6: Het opslaan van operanden in omgekeerde volgorde

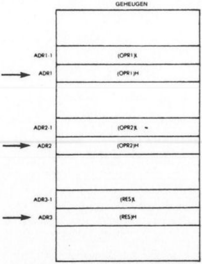


Fig. 3.7: Pointer naar het meest significante deel

Oefening 3.3: Neem aan, dat ADR1 niet wijst naar het minst significante deel van OP1 (zoals in figuren 3.5 en 3.6), maar naar het meest significante deel. Dit is in figuur 3.7 geillustreerd. Herschrijf het programma.

Het is de programmeur die beslist hoe de 16 bits getallen opgeslagen worden, en naar welk deel van het adres de adrespointers wijzen. Dat zijn allemaal keuzes die je moet maken als je algoritmes en datastructuren ontwerpt.

De hierboven geplaatste programma's zijn traditionele programma's: ze gebruiken de accumulator. We geven nu een alternatief programma, dat geen gebruik maakt van de accumulator, maar in plaats daarvan gebruik maakt van enkele speciale 16-bits instructies die beschikbaar zijn op de Z80. De operanden zijn opgeslagen zoals is aangegeven in figuur 3.5. Het programma luidt als volgt:

```
LD   HL,(ADR1)      OP1 IN HL
LD   BC,(ADR2)      OP2 IN BC
ADD  HL,BC          16-BITS OPTELLING
LD   (ADR3),HL      HL OPSLAAN OP ADR3
```

Merk op, dat dit een veel korter programma is dan het voorgaande. Het is "eleganter". *De Z80 staat toe, dat de registers H en L beperkt gebruikt kunnen worden als een 16-bits accumulator.*

Oefening 3.4: Neem aan dat 32-bits getallen opgeslagen zijn of opgeslagen moeten worden zoals dat is aangegeven in figuur 3.8. Schrijf dan een programma voor een 32-bits optelling, waarbij je gebruik maakt van de zojuist geintroduceerde instructies.

Antwoord:

```
LD   HL, (ADR1-1)
LD   BC, (ADR2-1)
ADD  HL, BC
LD   (ADR3-1), HL
LD   HL, (ADR1-3)
LD   BC, (ADR2-3)
ADC  HL, BC
LD   (ADR3-3), HL
```

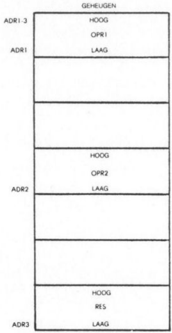


Fig. 3.8: Een 32-bits optelling

Nu we kunnen optellen, zullen we eens kijken naar aftrekken.

Het aftrekken van 16-bits getallen

We beginnen meteen met een 16-bits aftrekking, want een 8-bits aftrekking zou nu te eenvoudig voor ons zijn. OP1 en OP2 worden, zoals gebruikelijk, opgeslagen op de adressen ADR1 en ADR2. Het geheugen ziet eruit als getoond wordt in figuur 3.6. Het programma gebruikt de aftrek instructie (SBC) in plaats van ADD.

Oefening 3.5: Schrijf het aftrek programma.

Hier volgt het programma. Het datapad is te zien in figuur 3.9.

```
LD   HL, (ADR1)      OP1 IN HL
LD   DE, (ADR2)      OP2 IN DE
AND A                CARRYBIT WORDT 0
SBC  HL, DE          OP1 MIN OP2
LD   (ADR3), HL      RESULTAAT OP ADR3
```

Het programma lijkt op het programma voor de 16-bits optelling. Er zijn echter enkele verschillen. De Z80 kent twee instructies voor de optelling, n.l. ADD en ADC, maar slechts een instructie voor de aftrekking, SBC. Het eerste verschil tussen de programma's is het gebruik van SBC i.p.v. ADD. Het tweede verschil is de instructie AND A. Door deze instructie wordt het carrybit 0 gemaakt. De inhoud van A verandert niet.

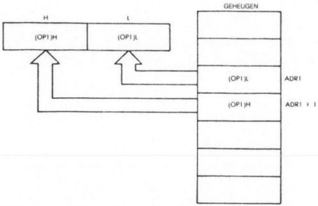


Fig. 3.9: 16 bit laden- LD HL, (ADR1)

Deze voorzorg is nodig, omdat de Z80 uitgerust is met twee soorten optelling: met en zonder carry bij het H en L register. Er is echter maar een soort aftrekking: SBC, aftrekken met carry. Omdat het carrybit altijd bij de aftrekking betrokken wordt, moet dit bit voor de aftrekking 0 gemaakt worden. Daarvoor hebben we de "AND A" instructie nodig.

Oefening 3.6: Herschrijf het aftrek programma, waarbij geen gebruik wordt gemaakt van de speciale 16-bits instructie.

Oefening 3.7: Schrijf een aftrek programma voor 8-bits getallen.

Men moet goed in gedachten houden, dat bij het rekenen in het twee-complement de uiteindelijke waarde van de carry geen betekenis heeft. Treedt een overflow op als gevolg van de aftrekking, dan wordt het overflowbit (V) in het status register 1 gemaakt. Dit bit kunnen we testen.

De getoonde voorbeelden zijn slechts eenvoudige binaire aftrekkingen en optellingen. Soms hebben we echter een andere rekenkunde nodig: de BCD rekenkunde.

REKENEN MET BCD

8-bits BCD optelling

De grondbeginselen van het BCD rekenen zijn behandeld in hoofdstuk 1. We zullen de hoofdpunten hier nog een keer herhalen. Het wordt hoofdzakelijk gebruikt voor administratieve toepassingen, waar het belangrijk is dat getallen tot op de laatste decimaal kloppen. Ieder cijfer wordt opgeslagen in een 4-bits nibble. Een byte kan dus twee BCD cijfers bevatten. Dit laatste heet *packed BCD*. Als voorbeeld tellen we twee bytes op

"01" wordt voorgesteld door 0000 0001
"02" wordt voorgesteld door 0000 0010

het resultaat is: 0000 0011

Dit is de BCD voorstelling voor "03". (Controleer dit zonodig door de tabellen aan het einde van dit boek te raadplegen.) In dit voorbeeld is alles nog eenvoudig. Maar laten we nu een ander voorbeeld bekijken.

"08" wordt voorgesteld door 0000 1000
"03" wordt voorgesteld door 0000 0011

Oefening 3.8: Bereken de som van bovenstaande optelling.

Een fout antwoord is "0000 1011", want dat is de *binaire* som van 8 en 3. "1011" is een in BCD niet voorkomende code en is dus verkeerd. Het antwoord is de BCD voorstelling van 11 decimaal, dus 0001 0001!

Dit probleem ontstaat, omdat de BCD voorstelling alleen de eerste 10 combinaties van de mogelijke 16 gebruikt. De 6 niet gebruikte combinaties komen in het BCD niet voor en zijn dus fout. "1011" is zo'n combinatie. Met andere woorden, als de som van twee binaire cijfers groter is dan 9 moeten we 6 bij het resultaat optellen om de 6 niet gebruikte combinaties over te slaan.

Tel de binaire voorstelling van 6 op bij 1011:

```
                          1011   (foute BCD code)
                        + 0110   (+6)
                     ---------
Het resultaat is:    0001 0001
```

Dit is de juiste BCD voorstelling van 11. We hebben dus nu het goede resultaat.

Dit voorbeeld laat een van de moeilijkheden van het rekenen met BCD zien. De 6 niet gebruikte codes moeten gecompenseerd worden. De Z80 beschikt daarvoor over een speciale instructie: "DAA". (tel 6 op als het resultaat groter is dan 9.)

Het voorbeeld laat nog een probleem zien. Het minst significante cijfer genereert een carry. Deze interne carry moet opgeteld worden bij het meest significante BCD cijfer. De optel instructie doet dit al automatisch, maar soms is het gemakkelijk te weten of er een carry is of niet. We kunnen dat detecteren m.b.v. het H bit in het status register. (De H komt van half-carry.)

Bij wijze van voorbeeld volgt nu een programma dat de BCD getallen 11 en 22 optelt:

```
LD   A, 11H        11 IN ACCUMULATOR
ADD A, 22H         TEL DAAR 22 BIJ OP
DAA                ZORG VOOR DE JUISTE BCD
                   REPRESENTATIE
LD   (ADR), A      SLA HET RESULTAAT OP
```

In het programma komt een nieuw symbool voor: "H". Deze H in het operand veld van de instructie geeft aan dat het getal ervoor een

hexadecimaal getal is. De getallen 0 tot en met 9 zijn voor de BCD en hexadecimale voorstelling gelijk. Het programma telt de getallen 11 en 22 op, en plaatst het resultaat op adres ADR. Als de operand is gespecificeerd als een onderdeel van de instructie, zoals in het voorbeeld, is er sprake van *"onmiddellijke" adressering*. De verschillende adresserings mogelijkheden komen in hoofdstuk 5 aan de orde.) Het opslaan van een resultaat op een adres zoals dat gebeurd in LD (ADR),A heet *absolute adressering*, als ADR een 16-bits adres voorstelt.

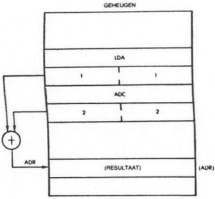


Fig. 3.10: Het opslaan van BCD cijfers

Het programma is gelijk aan de 8-bits optelling op de DAA instructie na. Wat de rol van deze instructie is zullen we zien in een voorbeeld:

Tel 11 en 22 op in BCD notatie.

$$\begin{array}{rl} 00010001 & (11) \\ +\ \underline{00100010} & (22) \\ =\ \underbrace{0011}_{3}\underbrace{0011}_{3} & (33) \end{array}$$

Gebruik makende van de regels van de binaire optelling komen we tot het juiste antwoord.

Tel nu 22 en 39 op en maak gebruik van de rekenregels voor de *binaire* optelling:

```
    00100010  (22)
  + 00111001  (39)
  ----------
  = 01011011
    5    ?
```

"1011" is een foute BCD code, omdat BCD slechts 10 van de mogelijke 16 codes gebruikt, en de laatste 6 codes overslaat. We moeten dus het zelfde doen, d.w.z. we tellen 6 bij het antwoord op:

```
    01011011  (binaire resultaat)
  +     0110  (6)
  ----------
  = 01100001  (61)
     6    1
```

Dit is het juiste BCD resultaat.

Oefening 3.9: Kan de instructie DAA in het programma ook achter LD (ADR),A geplaatst worden?

BCD aftrekking

De BCD aftrekking lijkt moeilijk. Om een aftrekking uit te voeren, moet het 10-complement opgeteld worden. In feite het zelfde principe als een binaire aftrekking door het twee-complement op te tellen. Het 10-complement wordt verkregen door het complement tot 9 te nemen en er dan 1 bij op te tellen. Normaal zou dat drie of vier instructies kosten. De Z80 is echter uitgerust met een zeer krachtige DAA instructie, welke automatisch het resultaat in de accumulator corrigeert, afhankelijk van de waarde van de C en H bits. (In het volgende hoofdstuk komen meer details van de DAA instructie aan de orde.)

16-bit BCD optelling

De 16-bits BCD optelling is net zo eenvoudig als de binaire optelling. Hier volgt meteen het programma:

```
LD   A,(ADR1)        OP1 LAAG IN A
LD   HL, ADR2        ADR2 IN HL
ADD A,(HL)           OP1 LAAG + OP2 LAAG
DAA                  CORRIGEER RESULTAAT
LD   (ADR3),A        SLA RESULTAAT LAAG OP
LD   A,(ADR1 + 1)    OP1 HOOG IN A
INC  HL              ADR1+1 IN HL
ADC  A,(HL)          OP1 HOOG + OP2 HOOG + CARRY
DAA                  CORRIGEER RESULTAAT HOOG
LD   (ADR3 + 1),A    SLA RESULTAAT HOOG OP
```

De packed BCD aftrekking

De elementaire BCD optelling en aftrekking is nu beschreven. In de praktijk kunnen BCD getallen echter ieder gewenst aantal cijfers bevatten. In het volgende vereenvoudigde programma van een BCD aftrekking nemen we aan dat twee getallen N1 en N2 het zelfde aantal bytes groot zijn. Het aantal bytes is COUNT. Figuur 3.11 geeft de geheugen indeling.

```
BCDPAK  LD   B,COUNT
        LD   DE, N2
        LD   HL, N1
        AND  A          MAAK CARRY 0
MINUS   LD   A,(DE)     N2 BYTE
        SBC  A,(HL)     N2 - 1
        DAA
        LD   (HL),A     SLA RESULTAAT OP
        INC  DE
        INC  HL
        DJNZ MINUS      TREK VAN B 1 AF,GA
                        NAAR MINUS ALS B NIET 0
```

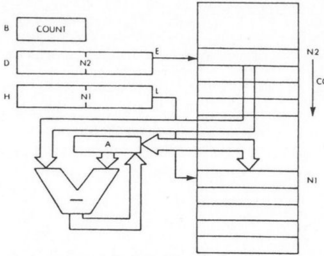


Fig. 3.11: Packed BCD aftrekking: N1 ← N2 − N1

N1 en N2 stellen de adressen voor, waar de BCD getallen zijn opgeslagen. Deze adressen komen in de register paren DE en HL:

```
BCDPAK LD    B, COUNT
       LD    DE, N2
       LD    HL, N1
```

Dan moet met het oog op de eerste aftrekking het carry bit 0 worden gemaakt. Dat kan op meerdere manieren worden gedaan. Hier gebeurt het met:

```
       AND    A
```

Het eerste byte van N2 gaat naar de accumulator, en het eerste byte van N1 wordt ervan afgetrokken. DAA corrigeert daarna het resultaat.

Het resultaat wordt opgeslagen in N1:

```
       LD     (HL),A
```

Tenslotte trekken we 1 af van de verwijsadressen of pointers:

```
INC   DE
INC   HL
```

Van B wordt 1 afgetrokken en de aftrek loop (een loop is een lus in het programma) wordt uitgevoerd totdat B gelijk aan 0 is:

```
DJNZ MINUS
```

DJNZ is een speciale Z80 instructie, die 1 aftrekt van B en een sprong maakt als B ongelijk aan 0 is.

Oefening 3.10: Vergelijk bovenstaand programma met dat van de binaire optelling. Wat is het verschil?

Oefening 3.11: Kunnen de rollen van DE en HL verwisseld worden? (Hint: wees voorzichtig met SBC).

Oefening 3.12: Schrijf een 16-bits BCD aftrek programma.

BCD status bits

Bij BCD optellingen geeft het carrybit aan, dat het resultaat groter is dan 99. Aangezien BCD in feite een echte binaire voorstelling is, is dit dus anders dan bij het twee-complement. Het carrybit geeft bij een aftrekking de borrow weer.

Instructie types

Tot nu toe hebben we twee soorten microprocessor instructies gebruikt. Allereerst LD, welke de accumulator laadt met data van een gegeven geheugenadres en omgekeerd. Dit is een *data verplaats* instructie.

Dan zijn er de *rekenkundige* instructies, zoals ADD, SUB, ADC en SBC. Ze voeren optellingen en aftrekkingen uit. Meer ALU instructies zullen nog in dit hoofdstuk worden behandeld.

Er zijn nog andere instructies beschikbaar, die we niet bekeken hebben. In het bijzonder zijn dat de "sprong" instructies, die de volgorde waarin het programma wordt doorlopen veranderen. Bij het volgende

voorbeeld zal dit soort instructie worden geintroduceërd. In gevallen waar een sprong (de Engelse term daarvoor is "jump") afhankelijk is van bepaalde logische condities, wordt deze wel "branch" of "vertakking" genoemd. Dit omdat de structuur van het programma met zijn vertakkingen wel op een boom lijkt.

VERMENIGVULDIGEN

We komen nu toe aan een meer ingewikkeld rekenkundig probleem: De vermenigvuldiging van binaire getallen. Alvorens het algoritme voor de binaire vermenigvuldiging te behandelen, bekijken we eerst een gewone decimale vermenigvuldiging:

```
     12  (vermenigvuldigtal)
  ×  23  (vermenigvuldiger)
  -----
     36  (deel produkt)
  + 24
  -----
  = 276  (uiteindelijk resultaat)
```

De vermenigvuldiging wordt uitgevoerd door het rechtse cijfer van de vermenigvuldiger te vermenigvuldigen met het vermenigvuldigtal, dus "3 × 12". Het deel produkt is 36. Daarna wordt 2 met 12 vermenigvuldigd. Het resultaat, 24, moet bij het deel resultaat worden opgeteld. Maar daarvoor moeten we eerst nog wat anders doen: 24 moet een plaats naar links geschoven worden. Hetzelfde resultaat wordt bereikt, door 36 een plaats naar rechts te verschuiven. Pas daarna kunnen de twee getallen worden opgeteld. Het uiteindelijke resultaat is 276. De binaire vermenigvuldiging wordt op precies de zelfde manier uitgevoerd. We zullen dat laten zien aan de hand van een voorbeeld. In dit voorbeeld wordt 5 met 3 vermenigvuldigd.

We bezien nu een voorbeeld. We vermenigvuldigen 5 × 3:

```
(5)        101  (VMT)
(3)    ×   011  (VMR)
         -----
           101  (DP)
          101
         000
         -----
(15)     01111  (RES)
```

De formele weergave van dit algorithme is te zien in figuur 3.12. We zullen dit stroomdiagram, ons eerste, eens nader bekijken.

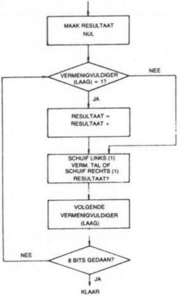


Fig. 3.12: Het basis vermenigvuldig algoritme. Stroomdiagram

Dit stroomdiagram is een symbolische weergave van het zojuist gepresenteerde algoritme. Iedere rechthoek stelt een uit te voeren opdracht voor. Het wordt vertaald in een of meer programma instructies. Iedere ruit is een test. Dit zijn dus de "vertakkingspunten" van het programma. Is de test goed, dan springen we naar een bepaald adres, is de test fout, dan springen we naar een ander adres. De principes van de sprong worden uitgelegd bij het programma zelf. De lezer moet zich er echter nu van verzekeren, dat dit stroomdiagram inderdaad het vermenigvuldig algoritme weergeeft. Merk op, dat er een pijl uit de onderste ruit komt, die naar de bovenste ruit gaat. Dat is gedaan, omdat hetzelfde deel van het stroomdiagram acht keer wordt uitgevoerd, een keer voor ieder bit van de vermenigvuldiger. Zo'n situatie, waar het programma vanaf een zelfde punt meerdere keren wordt doorlopen, heet een *programma lus* of een *loop*.

Oefening 3.13: Vermenigvuldig 4 met 7 binair, gebruik makende van het stroomdiagram. Controleer of het resultaat 28 is. Is dat niet het geval, doe de oefening dan opnieuw. Pas wanneer je het goede antwoord krijgt, ben je in staat het diagram te vertalen naar een programma.

8 bij 8 bits vermenigvuldiging

We gaan nu het stroomdiagram vertalen in een programma voor de Z80. Het complete programma is te vinden in figuur 3.13. Het zal tot in de details besproken worden. Zoals je je nog zult kunnen herinneren uit hoofdstuk 1 bestaat programmeren hier uit het vertalen van stroomdiagram 3.12 in het programma 3.13. Ieder blok in het diagram wordt vertaald in een of meer instructies.

Er wordt aangenomen, dat de vermenigvuldiger MPR en het vermenigvuldigtal MPD al een waarde hebben.

```
MPY88  LD   BC,(MPRAD)  VERMENIGV. IN C
       LD   B,8         B IS BIT TELLER
       LD   DE,(MPDAD)  VERM.TAL IN E
       LD   D,0         D WORDT 0
       LD   HL,0        RESULTAAT WORDT 0
                        GEMAAKT
MULT   SRL  C           SCHUIF VERM.BIT IN CARRY
       JR   NC,NOADD    TEST CARRY
       ADD  HL,DE       TEL MPD OP BIJ RESULTAAT
NOADD  SLA  E           SCHUIF MPD NAAR LINKS
       RL   D           RED BIT IN D
       DEC  B           TREK 1 AF VAN BIT TELLER
       JP   NZ,MULT     ALLES NOG EENS,ALS TELLER
                        NIET 0
       LD   (RESAD),HL  BERG RESULTAAT OP
```

Fig. 3.13: 8 × 8 vermenigvuldig programma

Het eerste blok in het diagram is nodig voor de *initialisatie*. Het maakt een aantal registers en geheugen locaties gelijk aan nul, als ze in het programma gebruikt worden. De registers die nodig zijn voor de vermenigvuldiging staan in figuur 3.14.

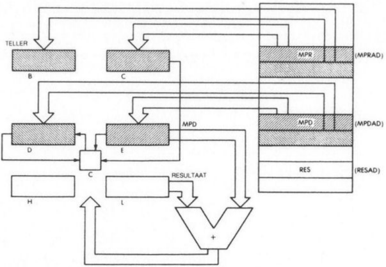


Fig. 3.14: 8 × 8 vermenigvuldiging. De registers

Er worden drie register paren gebruikt. MPRAD is het geheugen adres van de vermenigvuldiger. MPD is te vinden in het geheugen op adres MPDAD. De vermenigvuldiger en het vermenigvuldigtal worden respectievelijk in register C en E geladen (zie figuur 3.14). Register B wordt als teller gebruikt.

Registers D en E bevatten het vermenigvuldigtal als het een positie naar links wordt verschoven.

Merk op, dat hoewel in het begin alleen C en E geladen hoeven te worden, dit toch gebeurt met 16 bits tegelijk. B en D worden dus ook geladen. We moeten ze achteraf daarom de juiste waarde geven, B wordt 8 en D wordt 0.

Aangezien $2^8 \times 2^8 = 2^{16}$, kan het resultaat van een 8 bits maal 8 bits vermenigvuldiging 16 bits groot zijn. Er moeten daarom twee registers worden gereserveerd voor het resultaat. In ons geval de registers H en L. Zie figuur 3.14.

De eerste stap is de registers B, C en E te laden met de juiste inhoud, en het deel resultaat 0 te maken. Dat is te zien in stroomdiagram 3.12.

Alles wordt gedaan door de volgende instructies:

```
MPY88   LD   BC, (MPRAD)
        LD   B, 8
        LD   DE, (MPDAD)
        LD   D, 0
        LD   HL, 0
```

De eerste drie instructies laden respectivelijk MPR in register paar BC, de waarde "8" in register B, en MPD in register paar DE. Omdat MPR en MPD 8 bits woorden zijn worden ze eigenlijk in respectievelijk register C en E geladen, terwijl de woorden die in het geheugen volgen op MPR en MPD in B en D geladen worden. Dit wordt getoond in de figuren 3.15 en 3.16. De volgende instructie maakt de inhoud van D gelijk aan 0.

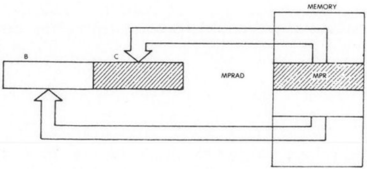


Fig. 3.15: LD BC, (MPRAD)

In dit programma wordt het vermenigvuldigtal naar links geschoven, voordat het wordt opgeteld bij het resultaat. (Het resultaat had voor de optelling ook naar rechts geschoven kunnen worden, zoals is aangegeven in het vierde blok van figuur 3.12.) Het vermenigvuldigtal wordt iedere stap in register D geschoven. Dit register moet daarom eerst nul gemaakt worden. Dat gebeurt in de vierde instructie. De vijfde instructie maakt tenslotte tegelijk de registers H en L gelijk aan nul.

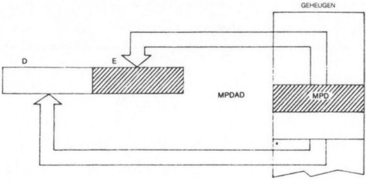


Fig. 3.16: LD DE, (MPDAD)

Kijken we weer naar het stroomdiagram, dan zien we, dat de volgende stap een test op het minst significante bit (het meest rechtse bit) van de vermenigvuldiger MPR is. Is dit bit een "1", dan moet de waarde van MPD opgeteld worden bij het deel resultaat, is dit bit "0", dan wordt er niet opgeteld. De volgende drie instructies doen dat:

```
MULT SRL  C
     JR   NC, NOADD
     ADD  HL, DE
```

Ons eerste probleem is hoe we het minst significante bit van de vermenigvuldiger kunnen testen. Dit zou kunnen met de BIT instructie van de Z80, waarmee ieder bit van ieder register getest kan worden. We willen echter hier het programma zo eenvoudig mogelijk houden, met een loop erin. Gebruikten we de BIT instructie, dan zouden we eerst bit 0 moeten testen, dan bit 1, enzovoort, tot aan bit 7. Daartoe zou iedere keer een andere instructie nodig zijn, en een eenvoudige loop zouden we niet kunnen toepassen. Om het programma te verkorten is hier een andere instructie gebruikt: de *verschuif* of *shift* instructie.

NB: Er is een manier om BIT te gebruiken in een loop, maar daarvoor zou het programma zichzelf moeten veranderen. En dat is iets, wat we graag willen vermijden.

SRL is een nieuw soort instructie van de ALU. Het is de afkorting van "shift right logical" (logische verschuiving naar rechts). Een logische verschuiving wordt gekenmerkt door het plaatsen van een 0 in bit 7. Dit in tegenstelling tot de rekenkundige verschuiving, waar de waarde die in bit 7 geschoven wordt gelijk is aan de vorige waarde van dat bit. We komen daar in het volgende hoofdstuk op terug. Het effect van SRL C wordt getoond in figuur 3.14: er komt een pijl uit register C, en gaat naar een vierkantje dat de carry voorstelt ("C"). Het meest rechtse bit van MPR komt terecht in de carry C, en wordt daar getest.

De volgende instructie, "JR NC,NOADD", is een *sprong* instructie. Het betekent "jump on no carry" (NC) naar adres (label) NOADD. (Jump on no carry = sprong als carry = 0). Als de inhoud van C "1" is, dan wordt er niet gesprongen, en de volgende instructie wordt uitgevoerd, d.w.z. "ADD HL,DE".

Deze instructie telt de inhoud van D en E op bij H en L; het resultaat komt in H en L. Aangezien E het vermenigvuldigtal bevat (zie figuur 3.14), wordt hierdoor het vermenigvuldigtal bij het deel resultaat opgeteld.

Nu wordt, ongeacht of MPD is opgeteld of niet, het vermenigvuldigtal naar links geschoven (dat is het vierde blok in het stroomdiagram):

```
NOADD SLA E
```

SLA is de afkorting van "shift left arithmetic" ofwel rekenkundige verschuiving naar links. We hebben net gezien, dat er twee soorten verschuivingen zijn: logische en rekenkundige. Onze instructie is van de laatste soort. Omdat het een verschuiving naar links is, wordt bit 0 gelijk aan 0 gemaakt.

Bij wijze van voorbeeld nemen we aan, dat de inhoud van E was 00001001. Na de SLA instructie wordt deze 00010010. De inhoud van het carrybit is dan 0.

We willen eigenlijk het meest significante bit (MSB) van E direct in D schuiven (zie figuur 3.14), maar aangezien er geen instructie is die dat kan, wordt het eerst in het carrybit geschoven. Daarna halen we het uit het carrybit en schuiven het in het D register:

```
RL D
```

RL is nog een ander soort instructie. Het betekent "roteer links". Bij de roteer instructie wordt het carry bit in het register geschoven, dit in tegenstelling tot de shift instructie, waar een bit het register uitgeschoven wordt in het carry bit. (Zie figuur 3.17.) En dat is precies wat we willen. De inhoud van het carry bit wordt in het rechtse bit van D geplaatst, en uiteindelijk hebben we dus een bit van E naar D geschoven.

Het effect van deze twee opeenvolgende instructies is geillustreerd in figuur 3.18. We zien, dat bit X op de meest significante plaats van E eerst wordt getransporteerd naar het carrybit. Daarna verhuist het naar de minst significante positie van D. Het effect is dus, dat we een bit van E naar D geschoven hebben.

Volgens het stroomdiagram moeten we nu naar het volgende bit van MPR wijzen, en controleren of dit het achtste bit is. Dat wordt gedaan door van de byte teller 1 af te trekken. Register B bevat deze teller. (Zie figuur 3.14.)

DEC B

Dit is een "decrement" instructie, wat zeggen wil, dat van de inhoud van B de waarde 1 wordt afgetrokken.

Tenslotte moeten we controleren of de teller al 0 is. Daartoe wordt het Z bit getest. Je zult je ongetwijfeld nog herinneren, dat het Z(ero) status bit aangeeft of de uitkomst van de vorige rekenkundige instruc-

JP NZ,MULT

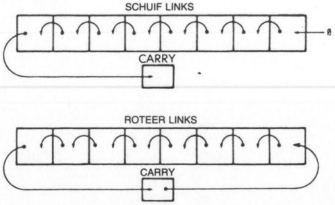


Fig. 3.17: Shift (schuiven) en roteren

tie de waarde 0 had. We moeten hier opmerken, dat de instructies DEC HL, DEC BC, DEC DE, DEC IX en DEC SP het Z bit niet beinvloeden. Als de teller niet "0" is, moet de programma loop nogmaals doorlopen worden. Dat wordt bereikt door de volgende instructie:

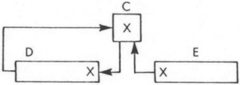


Fig. 3.18: Schuiven van E naar D

Dit is een sprong instructie (JP is de afkorting van jump), welke specificeert, dat als Z gelijk is aan "0" (NZ betekent non-zero), er gesprongen wordt naar het adres MULT. Het gevolg is een programma loop, die doorlopen wordt totdat B de waarde 0 krijgt. Is dat het geval (B = 0), dan wordt niet gesprongen, en wordt de volgende instructie uitgevoerd:

LD (RESAD),HL

Deze instructie redt de inhoud van het register paar HL, dus het resultaat van de vermenigvuldiging, op adres RESAD. D.w.z. de inhoud van H gaat naar adres RESAD, en de inhoud van L naar RESAD + 1. Deze instructie redt dus 16 bits tegelijk.

Oefening 3.14: Kun je nu hetzelfde vermenigvuldigings programma schrijven, maar dan door de BIT instructie (wordt beschreven in het volgende hoofdstuk) te gebruiken i.p.v. de SRL C instructie? Wat zijn de nadelen?

We gaan, als dat mogelijk is, het programma verbeteren:

Oefening 3.15: Kan JP aan het einde van het programma vervangen worden door JR? Als dat mogelijk is, wat is dan het voordeel?

Oefening 3.16: Kan DJNZ gebruikt worden om het programma korter te maken?

Oefening 3.17: Bekijk de instructies LD D,0 en LD HL,0 aan het begin van het programma. Kunnen ze vervangen worden door:

```
XOR   A
LD    D, A
LD    H, A
LD    L, A
```

Zoja, wat is dan de verbetering, gemeten in het aantal bytes en snelheid?

NB: In de meeste gevallen zal het zojuist ontwikkelde programma een subroutine zijn, en is de laatste instructie RET (return). Dat mechanisme wordt later in dit hoofdstuk besproken.

Test je zelf (zeer belangrijk)

We hebben nu het eerste echte programma doorgenomen. We zijn vele verschillende soorten instructies tegengekomen, zoals: verplaats instructies (LD), rekenkundige instructies (ADD), logische instructies (SRL, SLA, RL) en sprong instructies (JR, JP). Zelfs een loop behoorde tot het programma, waarin de laatste zeven instructies, beginnende op adres MULT, herhaald uitgevoerd werden. Om het programmeren te leren, is het noodzakelijk dergelijke programma's tot in de details te begrijpen. Dit programma is veel langer dan de programma's daarvoor, en het zou gedetailleerd bestudeerd moeten worden. We stellen daarom de volgende oefening voor. Maak de oefening helemaal af en ga niet verder tot je de goede uitkomst hebt. Want alleen een goed antwoord is het bewijs dat je de tot nu toe behandelde principes begrijpt. Wie de oefening niet maakt, of toch doorgaat na een fout antwoord, zal zeker moeilijkheden ondervinden bij het schrijven van programma's later. Oefening baart kunst, ook bij het programmeren. Neem nu even een pauze en maak daarna de volgende oefening:

Oefening 3.18: Iedere keer als een programma is geschreven, zou het met de hand gecontroleerd moeten worden, om er zeker van te zijn, dat de resultaten correct zijn. Dat is precies wat we nu gaan doen: het doel van deze oefening is de tabel van figuur 3.19 volledig en nauwkeurig in te vullen.

LABEL	INSTRUCTIE	B	C	C (CARRY)	D	E	H	L

Fig. 3.19: Formulier voor de vermenigvuldig oefening

Je kunt meteen de tabel in het boek invullen, maar als je het boek niet wilt beschadigen kun je het eerst copieren. Doel is de inhouden van de belangrijke registers na iedere instructie in de tabel in te vullen. De registers die het programma gebruikt zijn, van links naar rechts: B en C, de carry C, de registers D en E, en als laatsten H en L. Links in de tabel moeten ingevuld worden: de labels, als deze aanwezig zijn, en de instructies. In de dan volgende kolommen moeten de inhouden van de

registers ingevuld worden. Zet een streepje in de kolom als de inhoud niet bekend of onbepaald is. Laten we samen beginnen het formulier in te vullen. Daarna zul je het alleen af moeten maken. De eerste regel is als volgt:

LABEL	INSTRUCTIE	B	C	C	D	E	H	L
		--	--	-	--	--	--	--
MP488	LD BC,(0200)	00	03	-	--	--	--	--

Fig. 3.20: Vermenigvuldiging: na een instructie

We nemen aan, dat we 3 (MPR) met 5 (MPD) vermenigvuldigen.

De eerste instructie is "LD BC,(MPRAD)". De inhoud van adres MPRAD wordt in registers B en C geladen. MPR is 3, dus: "00000011". Na de instructie is de inhoud van C dus 3. B wordt geladen met de inhoud van MPRAD + 1, welke onbepaald is. De volgende instructie zorgt er voor dat B de waarde 8 krijgt. Zie figuur 3.21. De inhouden van D, E, H en L zijn nog niet bekend, dat worden dus streepjes. De inhoud van het carry bit is ook nog onbepaald, aangezien de LD instructie dat bit niet beinvloedt. Ook hier dus een streepje.

LABEL	INSTRUCTIE	B	C	C	D	E	H	L
		--	--	-	--	--	--	--
MP488	LD BC,(0200)	00	03	-	--	--	--	--
	LD B,08	08	03	-	--	--	--	--

Fig. 3.21: Vermenigvuldiging: na twee instructies

Figuur 3.22 geeft de situatie na vijf instructies (tot aan MULT).

LABEL	INSTRUCTIE	B	C	C	D	E	H	L
		--	--	-	--	--	--	--
MP488	LD BC,(0200)	00	03	-	--	--	--	--
	LD B, 08	08	03	-	--	--	--	--
	LD DE,(0202)	08	03	-	00	05	--	--
	LD D, 00	08	03	-	00	05	--	--
	LD HL,0000	08	03	-	00	05	00	00

Fig. 3.22: Vermenigvuldiging: na vijf instructies

De SRL instructie veroorzaakt een logische verschuiving naar rechts, en het rechtse bit van MPR komt in het carry bit terecht. In figuur 3.23 is te zien, dat de inhoud van MPR na de verschuiving gelijk is aan "00000001". Carry C is dus 1. Alle andere registers worden niet veranderd door deze instructie. Ga nu verder met het invullen.

LABEL	INSTRUCTIE	B	C	C	D	E	H	L
		--	--	-	--	--	--	--
MP488	LD BC,(0200)	00	03	-	--	--	--	--
	LD B, 08	08	03	-	--	--	--	--
	LD DE,(0202)	08	03	-	00	05	--	--
	LD D, 00	08	03	-	00	05	--	--
	LD HL,0000	08	03	-	00	05	00	00
MULT	SRL C	08	01	1	00	05	00	00
	JR NC,0114	08	01	1	00	05	00	00
	ADD HL,DE	08	01	0	00	05	00	05
NOADD	SLA E	08	01	0	00	0A	00	05
	RL D	08	01	0	00	0A	00	05
	DEC B	07	01	0	00	0A	00	05
	JP NZ,010F	07	01	0	00	0A	00	05

Fig. 3.23: Na de loop een maal doorlopen te hebben

De compleet ingevulde tabel staat in figuur 3.39 aan het eind van dit hoofdstuk. Figuur 3.40 geeft de hex of decimale listing van het programma.

Programmeer alternatieven

Het net door ons ontwikkelde programma zou op vele andere manieren geschreven kunnen worden. In het algemeen geldt, dat iedere programmeur zijn eigen manieren heeft om een programma te veranderen, en zelfs te verbeteren. Zo hebben wij, bijvoorbeeld, het vermenigvuldigtal naar links geschoven. Wiskundig is het net zo juist het resultaat een positie naar rechts te schuiven voor het op te tellen bij het vermenigvuldigtal. Eigenlijk is dat best een interessante oefening!

Oefening 3.19: Schrijf een 8 × 8 vermenigvuldig programma, gebruik makende van het zelfde algoritme. Schuif nu echter het resultaat een positie naar rechts, i.p.v. het vermenigvuldigtal een positie naar links te schuiven. Vergelijk dit programma met het vorige. Is het nieuwe programma sneller of langzamer?

Een verbeterd vermenigvuldig programma

Het zojuist gemaakte programma is een recht-toe-recht-aan vertaling van het algoritme. Om echter een efficient programma te kunnen maken, moeten we op de details letten. Vaak kan de lengte van het programma korter worden. Meestal gaat dat gepaard met een hogere uitvoer snelheid. We zullen proberen ons basis programma te verbeteren.

Stap 1

Een eerste verbetering is een efficienter gebruik van de Z80 instructie set. De op een na laatste instructie en de instructie daarvoor kunnen worden vervangen door een enkele instructie:

```
DJNZ MULT
```

Dit is een "automatische sprong": B wordt met 1 verlaagd, en er wordt gesprongen naar de gespecificeerde locatie als B ongelijk 0 is. De instructie is niet helemaal gelijk aan het vorige paar:

```
DEC B
JP   NZ, MULT
```

De instructie geeft namelijk een *verplaatsing*, en die verplaatsing mag niet kleiner zijn dan −256 of groter dan +256. In het programma wordt maar een paar bytes verder gesprongen, dus deze verbetering is toegestaan. Het programma wordt, zoals dat is weergegeven in figuur 3.24.

```
MP488B  LD    DE, (MPDAD)
        LD    BC, (MPRAD)
        LD    B, 8              BIT TELLER
        LD    HL, 0
MULT    SRL   C
        JR    NC, NOADD
        ADD   HL, DE
NOADD   SLA   E
        RL    D
        DJNZ  MULT
        LD    (RESAD), HL
        RET
```

Fig. 3.24: Verbeterde vermenigvuldiging. Stap 1

Stap 2

Om het programma nog meer te kunnen verbeteren kijken we naar de twee verschillende schuif instructies die gebruikt worden in het programma van figuur 3.13. De vermenigvuldiger wordt naar rechts geschoven, het vermenigvuldigtal daarna naar links. Eerst wordt register E naar links geschoven, waarna register D naar rechts wordt geroteerd. Dat alles is nogal tijdrovend. Een standaard programmeer "truc", die gebruikt wordt bij vermenigvuldigingen is gebaseerd op het volgende: iedere keer als de vermenigvuldiger een positie wordt verschoven, komt een ander bit in het vermenigvuldiger register beschikbaar. Bijvoorbeeld: als de vermenigvuldiger naar rechts schuift, komt links een

bit vrij. Tegelijk zien we, dat het allereerste deel product ("resultaat") hooguit 9 bits groot is. Als we aan het begin van het programma een enkel register aanwijzen voor het resultaat, kunnen we het vrijgekomen bit bij de vermenigvuldiger gebruiken voor de rest van het resultaat. Na de volgende verschuiving van MPR wordt het deel product een bit groter. Ook dit bit plaatsen we op de vrijgekomen positie bij de vermenigvuldiger, enz.

Het programma kan dus verbeterd worden, door MPR en RES in een register paar te plaatsen. Ideaal zou het zijn, als we ze ook nog tegelijk zouden kunnen verschuiven. Helaas kan de Z80 slechts 8-bits registers verschuiven. Net als de meeste 8-bits microprocessors heeft de Z80 geen instructie waarmee 16 bits tegelijk worden verschoven.

Maar ook hier kunnen we een truc gebruiken. De Z80 (en ook de 8080) heeft een 16-bits optel instructie. We hebben deze instructie al gebruikt. Neem aan, dat de vermenigvuldiger en het resultaat in H en L zitten. De volgende instructie kan dan gebruikt worden:

```
ADD HL,HL
```

H en L worden bij zichzelf opgeteld. H en L worden dus verdubbeld. In het binaire systeem komt verdubbeling overeen met een verschuiving van een bit naar links. We hebben dus een 16-bits verschuiving gereali-

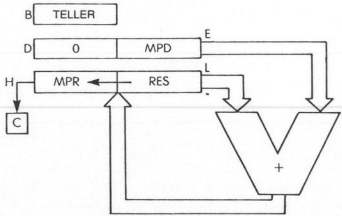


Fig. 3.25: De registers bij de verbeterde vermenigvuldiging

seerd d.m.v. een instructie. Helaas is het een verschuiving naar links, terwijl deze naar rechts had moeten zijn. Maar dat is helemaal geen probleem.

Principieel kan MPR zowel naar links als naar rechts geschoven worden. Wij hebben de verschuiving naar rechts gebruikt, omdat dat bij een gewone optelling ook wordt gedaan. Het hoeft echter niet zo te gebeuren. Een verschuiving van de MPR naar links is even juist.

Om het voordeel van de gesimuleerde 16-bits verschuiving te kunnen gebruiken, moet de MPR naar links geschoven worden. MPR komt dus in register H en het resultaat in register L. Figuur 3.25 geeft de aldus ontstane situatie.

De rest van het programma blijft zo ongeveer gelijk aan het vorige programma:

```
MUL88C  LD    HL,(MPRAD-1)
        LD    L, 0
        LD    DE,(MPDAD)
        LD    D, 0           TELLER
        LD    B, 8           VERSCHUIVING
MULT    ADD   HL, HL         NAAR LINKS
        JR    NC,NOADD
        ADD   HL,DE
NOADD   DJNZ  MULT
        LD    (RESAD),HL
        RET
```

Fig. 3.26: Verbeterde vermenigvuldiging. Stap 2.

Bij een vergelijking met het vorige programma valt op, dat de lengte van de vermenigvuldigings loop (de instructies tussen MULT en de sprong) kleiner is geworden. Het programma heeft minder instructies, en meestal betekent dat een grotere snelheid waarmee het programma wordt uitgevoerd. Een juiste keuze van de registers kan dus voordelig zijn.

Een recht-toe-recht-aan ontwerp resulteert in het algemeen in een programma dat werkt. Vrijwel nooit is dat programma *optimaal*. Het is daarom zo belangrijk de beschikbare registers en instructies te begrijpen en op de best mogelijke manier te gebruiken.

Oefening 3.20: Bereken de snelheid van een vermenigvuldiging m.b.v. het laatste programma. Neem aan, dat in 50% van de gevallen gesprongen wordt. Het aantal cycli per instructie is te vinden in het volgende hoofdstuk. De klokfrequentie is 2 MHz (1 klokcyclus = 0,5 microseconde).

Oefening 3.21: In het voorbeeld is register paar D en E gebruikt voor het vermenigvuldigtal. Hoe moet het programma veranderd worden, als we B en C willen gebruiken? (Hint: aan het eind moet iets veranderd worden.)

Oefening 3.22: Waarom moet register D nul worden gemaakt als MPD in E wordt geladen?

Als laatste kijken we naar een detail, dat de niet met de Z80 vertrouwde programmeur misschien irriteert. De lezer heeft misschien al opgemerkt, dat bij het laden van MPD in register E, zowel register D als E geladen werden uit het geheugen. Dat gebeurt, omdat, als niet eerst het adres in H en L wordt geplaatst, het niet mogelijk is een enkel byte uit het geheugen te halen. Dat is de erfenis van de 8008, die geen

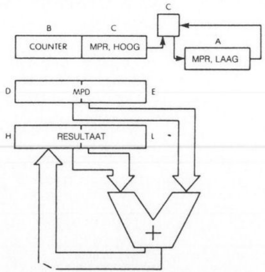


Fig. 3.27: 16 × 16 vermenigvuldiging. De registers.

directe adresseringsmogelijkheid had. Dat is overgenomen door de 8080, met enkele verbeteringen, en later ook door de Z80. Daar is het mogelijk 16 bits direct uit het geheugen te halen, maar geen 8 bits.

Nu we dit mogelijke mysterie hebben opgelost, gaan we door met een ingewikkelder vermenigvuldiging.

Een 16 × 16 vermenigvuldiging

Om onze net verworven vaardigheden te testen, vermenigvuldigen we twee 16-bits getallen. We nemen echter aan dat het resultaat niet meer dan 16 bits groot is, zodat het in een register paar past.

Het resultaat van de eerste vermenigvuldiging stond in registers H en L (zie figuur 3.27), terwijl de registers D en E MPD bevatten.

Het is aanlokkelijk B en C te gebruiken voor de vermenigvuldiger. Als we echter voordeel willen hebben van de DJNZ instructie, moeten we B gebruiken als teller. Als gevolg daarvan stoppen we een helft van de vermenigvuldiger in C, en de andere helft in A (zie figuur 3.27). Het programma volgt nu:

```
MUL16  LD    A, (MPRAD + 1)   MPR,HOOG
       LD    C,A
       LD    A, (MPRAD)       MPR,LAAG
       LD    B, 16            TELLER
       LD    DE, (MPDAD)      MPD
       LD    HL,0
MULT   SRL   C                VERSCHUIF RECHTS
                              MPR,HOOG
       RRA                    ROTEER RECHTS
                              MPR,LAAG
       JR    NC, NOADD        TEST CARRY
       ADD   HL, DE           TEL MPD OP BIJ
                              RESULTAAT
NOADD  EX    DE,HL
       ADD   HL, HL           VERSCHUIF MPD
                              NAAR LINKS
       EX    DE,HL
       DJNZ  MULT
       RET
```

Fig. 3.28: 16 × 16 vermenigvuldig programma

Dit programma is analoog aan de voorgaande programma's. De eerste zes instructies (MUL16 tot MULT) initialiseren de registers met de juiste inhoud. Een moeilijkheid vormt het feit, dat MPR geladen moet worden d.m.v twee bewerkingen. MPRAD wijst naar het minst significante deel van MPD in het geheugen. Het meest significante deel bevindt zich op het volgende geheugenadres. (Dit zou ook andersom kunnen zijn.) Als het meest significante deel in A is geplaatst, moet het verhuizen naar C:

```
LD      A, (MPRAD + 1)
LD      C, A
```

Tenslotte wordt het minst significante deel in de accumulator geplaatst:

```
LD      A, (MPRAD)
```

De andere registers (B, D, E, H en L) worden op de gewone manier geinitialiseerd:

```
LD      B, 16
LD      DE, (MPDAD)
LD      HL, 0
```

Er moet een 16 bits verschuiving worden uitgevoerd op de vermenigvuldiger. Daarvoor zijn twee afzonderlijke schuif of roteer instructies nodig, een voor A en een voor C:

```
MULT    SRL     C
        RRA
```

Na de verschuiving bevat het carrybit het rechtse en dus minst significante bit van MPR. Daar wordt het getest:

```
JR      NC, NOADD
```

Het vermenigvuldigtal wordt wel opgeteld als C op 1 staat en niet opgeteld als C op 0 staat:

```
ADD     HL, DE
```

Als volgende stap moet MPD een positie naar links worden verschoven. De Z80 bezit echter geen instructie die D en E tegelijk verschuift. Ook kan de Z80 de inhoud van DE bij zichzelf optellen. Daarom wordt de inhoud van D en E in H en L geplaatst, waar het verdubbeld wordt. Daarna gaat de nieuwe inhoud weer terug naar D en E:

```
NOADD   EX      DE,HL
        ADD     HL,HL
        EX      DE,HL
```

Tenslotte moet B met 1 worden verminderd. Er wordt een sprong gemaakt, als B niet gelijk is aan 0:

```
        DJNZ  MULT
```

Ook hier kunnen andere registers gekozen worden, die misschien resulteren (of misschien ook niet) in een korter programma:

Oefening 3.23: Plaats de vermenigvuldiger in de registers B en C. Schrijf het programma en bespreek de voor- en nadelen van deze register keuze.

Oefening 3.24: Bekijk het programma in figuur 3.28. Kun je een andere manier bedenken om MPD te verschuiven, zonder het eerst naar HL te verplaatsen?

Oefening 3.25: Schrijf een 16 × 16 vermenigvuldig programma, dat ontdekt dat het resultaat groter dan 16 bits is. Dit vormt een eenvoudige verbetering van het basis programma.

Oefening 3.26: Schrijf een 16 × 16 vermenigvuldig programma met een 32 bits groot resultaat. Figuur 3.29 geeft een voorstel voor de toekenning van de registers. Onthoud, dat het eerste deelresultaat slechts 16 bits groot is, en dat na iedere stap een bit vrij komt bij de vermenigvuldiger.

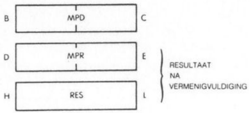


Fig. 3.29: 16 × 16 vermenigvuldiging met 32-bits resultaat

Als laatste rekenkundige bewerking bespreken we de deling.

BINAIRE DELING

Het algoritme voor de binaire deling lijkt op het algoritme voor de vermenigvuldiging. De deler wordt iedere keer afgetrokken van de hogere orde bits van het deeltal. Na iedere aftrekking wordt het resultaat gebruikt i.p.v. het oorspronkelijke deeltal. De waarde van het quotient wordt bij iedere aftrekking met 1 verhoogd. Na verloop van tijd wordt het resultaat van de aftrekking negatief. Dan moet het deelresultaat hersteld worden, door de deler er weer bij op te tellen. Bij het quotient moet ook weer 1 worden opgeteld. Quotient en deeltal worden een positie naar links geschoven en het algoritme wordt herhaald. Het stroomdiagram is te vinden in figuur 3.30.

16 × 8 deling

Als voorbeeld behandelen we hier een 16 × 8 deling, met een 8-bits quotient en een 8-bits rest. De toewijzing van de registers staat in figuur 3.31.

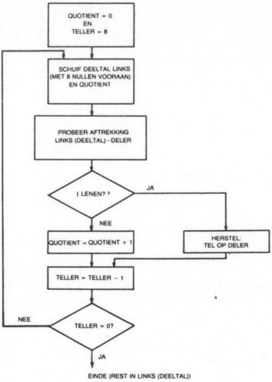


Fig. 3.30: 8 bits binaire deling. Stroomdiagram.

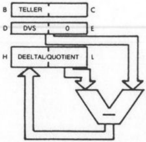


Fig. 3.31: 16 × 8 deling. De registers.

Het programa is als volgt:

```
DIV168  LD    A, (DVSAD)   LAAD DELER
        LD    D, A         IN A
        LD    E,0
        LD    HL, (DVDAD)  LAAD 16-BITS DEELTAL
        LD    B, 8         INITIALISEER TELLER
DIV     XOR   A            CARRY WORDT 0
        SBC   HL, DE       DEELTAL-DELER
        INC   HL           QUOTIENT=QUOTIENT+1
        JP    P, NOADD     TEST OF REST POSITIEF IS
        ADD   HL, DE       HERSTEL INDIEN NODIG
        DEC   HL           QUOTIENT=QUOTIENT−1
NOADD   ADD   HL, HL       SCHUIF DEELTAL NAAR
                           LINKS
        DJNZ  DIV          LOOP TOT B = 0
        RET
```

Fig. 3.32: 16 × 8 deling. Programma.

De eerste vijf instructies laden de deler en het deeltal in de goede registers. De teller wordt geinitialiseerd (B krijgt de waarde 8). Ook hier is B de aangewezen positie voor de teller als de DJNZ instructie wordt gebruikt.

```
DIV168    LD  A,(DVSAD)
          LD  D,A
          LD  E,0
          LD  HL,(DVDAD)
          LD  B,8
```

Dan wordt de deler afgetrokken van het deeltal. Omdat er geen aftrek instructie is zonder carry, moet deze eerst 0 worden gemaakt. Dat kan op vele manieren. Bijvoorbeeld m.b.v. de volgende instructies:

```
XOR   A
AND   A
OR    A
```

In ons programma gebruiken we:

```
DIV     XOR     A
```

Dan volgt de aftrekking:

```
        SBC     HL,DE
```

Aangenomen wordt dat de aftrekking succesvol is, d.w.z. dat de rest positief is. Daarom wordt bij het quotient 1 opgeteld. Als de aftrekking fout gaat (de rest is negatief) Moeten we weer 1 aftrekken van het quotient:

```
        INC     HL
```

Het resultaat van de aftrekking wordt getest:

```
        JP      P,NOADD
```

Is de rest groter of gelijk aan 0, dan is de aftrekking juist geweest, en hoeven we niets op te slaan. Het programma springt naar adres NOADD. Is de aftrekking fout, dan moet het deeltal de oude waarde weer krijgen, en van het quotient moet 1 worden afgetrokken:

```
        ADD     HL,DE
        DEC     HL
```

Tenslotte moet het overblijvende deeltal naar links worden geschoven, vooruit lopend op de volgende aftrekking. B (de teller) wordt met 1 verminderd, en getest op de waarde 0. De loop wordt doorlopen zolang B ongelijk aan 0 is:

```
NOADD   ADD     HL,HL
        DJNZ    DIV
        RET
```

Oefening 3.27: Controleer de werking van de deling door figuur 3.33 in te vullen, zoals we dat ook hebben gedaan bij oefening 3.18. Merk op dat D niet op het formulier voor hoeft te komen omdat de inhoud ervan niet verandert.

LABEL	INSTRUCTIE	B	H	L

Fig. 3.33: Formulier voor de deling

8-bits deling

Het volgende programma gebruikt, net als het voorgaande programma, de "bewaar methode". In A komt het gecomplementeerde quotiënt. Het is een 8 × 8 bits deling (zonder teken).

```
        ;E IS DEELTAL
        ;C IS DELER
        ;A IS QUOTIENT
        ;B IS REST

DIV88   XOR    A         ACCU WORDT 0
        LD     B,8       LOOP TELLER
LOOP88  RL     E         ROTEER CARRY IN DEEL-
                         TAL.
        RLA              CARRY WORDT 0
        SUB    C         TREK DELER AF
        JR     NC,$ + 3  AFTREKKING OK
        ADD    A,C       ACCU OUDE WAAR-
                         DE,CARRY 1
        DJNZ   LOOP88
        LD     B,A       REST IN B
        LD     A,E       HAAL QUOTIENT
        RLA              SCHUIF LAATSTE RES.BIT
        CPL              COMPLEMENTEER BITS
        RET
```

NB: $ is het symbool voor de waarde van de programma teller.

Deling volgens de "niet bewaar" methode

Het volgende programma voert een 16-bits door 15-bits deling uit d.m.v de niet bewaar methode. IX wijst naar het deeltal, IY naar de deler (niet nul). Zie figuur 3.34.

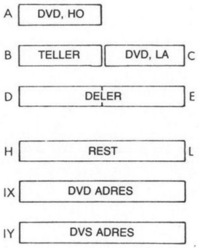


Fig. 3.34: Deling volgens niet bewaar methode. De registers.

Register B is teller, beginwaarde 16.
A en C bevatten het deeltal.
D en E bevatten de deler.
H en L bevatten het resultaat.
Het 16-bits deeltal wordt verschoven door:

```
RL    C
RLA
```

De rest wordt verschoven door:

```
ADC  HL,HL
```

```
DIV16    LD    B, (IX + 1)
         LD    C, (IX)
         LD    D,(IY + 1)
         LD    E, (IY)
         LD    A, D
         OR    E          (DELER) HOOG of
                          (DELER) LAAG
         JR    Z, ERROR   TEST DELER = 0
         LD    A, B       HAAL (DEELTAL) HOOG
         LD    HL, 0      RESULTAAT WORDT 0
         LD    B, 16      TELLER
TRIALSB  RL    C          ROTEER RES. EN ACCU
                          LINKS
         RLA
         ADC   HL, HL     SCHUIF LINKS. C BLIJFT 0
         SBC   HL, DE     MINUS DELER
NULL     CCF              RESULTAAT BIT
         JR    NC, NGV    ACCU NEGATIEF?
PTV      DJNZ  TRIALSB    TELLER 0?
         JP    DONE
RESTOR   RL    C          ROTEER RES. + ACCU
                          LINKS
         RLA
         ADC   HL, HL     ZIE BOVEN.
         AND   A
         ADC   HL, DE     HERSTEL DOOR DELER
                          OP TE TELLEN
         JR    C, PTV     RES. POSITIEF
         JR    Z, NULL    RES. NUL
NGV      DJNZ  RESTOR     TELLER 0?
DONE     RL    C          SCHUIF RES. BIT BINNEN
         RLA
         ADD   HL, DE     CORRIGEER REST
         LD    B, A       QUOTIENT IN B, C
         RET
```

Het uiteindelijke quotient komt in B en C, en de rest in HL. Het programma volgt nu:

Oefening 3.28: Vergelijk het vorige programma met het nu volgende. Dit programma gebruikt de bewaar techniek.

```
        DEELTAL IN AC.
        DELER IN DE
        QUOTIENT IN AC
        REST IN HL

DIV16   LD      HL,0     ;MAAK ACCU NUL
        LD      B,16     ;SET TELLER
LOOP16  RL      C        ;ROTEER RESULTAAT EN
        RLA              ;ACCUMULATOR LINKS
        ADC     HL,HL    ;VERSCHUIVING NAAR
                         ;LINKS
        SBC     HL,DE    ;TREK DELER AF
        JR      NC,$ + 3 ;AFTREKKING OK
        ADD     HL,DE    ;HERSTEL ACCUMULATOR
        CCF              ;BEREKEN RESULTAAT BIT
        DJNZ    LOOP16   ;TELLER NIET NUL
        RL      C        ;HAAL LAATSTE RES.BIT
                         ;BINNEN
        RET
        RLA
```

NB: $ betekent huidige adres (zevende instructie).

LOGISCHE BEWERKINGEN

De andere klasse van instructies die uitgevoerd kunnen worden door de ALU in de microprocessor zijn de logische instructies. Hiertoe behoren: AND (EN), OR (OF) en exclusieve OR of XOR. Bovendien kunnen we hier ook de schuif en roteer instructies en de vergelijk instructie toe te rekenen. Het gebruik van AND, OR en XOR wordt in hoofdstuk 4 besproken.

Nu gaan we een klein programma maken, dat test of een bepaald geheugen adres LOC de waarde 0, de waarde 1 of iets anders bevat.

In dit programma wordt de vergelijk instructie geintroduceerd,

waarmee een serie logische tests wordt uitgevoerd. Afhankelijk van de uitkomst van de vergelijking wordt een bepaald programma deel uitgevoerd.
Hier volgt het programma:

```
         LD    A,(LOC)     LEES KARAKTER IN LOC
         CP    00H         VERGELIJK MET NUL
         JP    Z,ZERO      IS HET NUL?
         CP    01H         VERGELIJK MET EEN
         JP    Z,ONE
NONFND   ...
         ...
ZERO     ...
         ...
ONE      ...
```

De eerste instructie: "LD A,(LOC)" leest de inhoud van geheugen adres LOC, en laadt die in de accumulator. Dat is het karakter dat we willen testen. De volgende instructie vergelijkt het met de waarde 0:

```
CP    00H
```

Deze instructie vergelijkt de inhoud van de accumulator met de hexadecimale waarde "00", d.w.z. "00000000". Zijn de waarden gelijk, dan wordt het Z bit in het status register 1 gemaakt. Dat bit kan getest worden met de volgende instructie:

```
JP    Z, ZERO
```

Als Z de waarde 1 heeft, wordt gesprongen naar adres ZERO. Als Z de waarde 0 heeft, de accumulator is dus ongelijk aan 0, dan wordt de volgende instructie uitgevoerd:

```
CP    01H
```

Op gelijke wijze als hierboven wordt naar ONE gesprongen als de accumulator de waarde 1 heeft. Is de waarde niet gelijk aan een van de waarden waarmee vergeleken is, dan wordt de instructie achter label NONFND uitgevoerd.

```
         JP     Z, ONE
NONFND   ...
```

Met dit programma is de waarde van de vergelijk instructie gevolgd door een sprong gedemonstreerd. Deze combinatie zal nog veel in onze programma's gebruikt worden.

Oefening 3.29: Onderzoek m.b.v. de beschrijving in het volgende hoofdstuk de instructie LDA, (LOC). Wat is de invloed op de vlaggen? Is de volgende instructie (CP 00H) wel nodig?

Oefening 3.30: Schrijf een programma dat de inhoud van geheugen adres "24" leest en springt naar adres "STAR", als de inhoud een "*" is. De code voor "*" is "00101010".

SAMENVATTING VAN DE INSTRUCTIES

De meeste belangrijke instructies van de Z80 hebben we leren kennen door ze te gebruiken. We hebben data tussen het geheugen en registers getransporteerd. We hebben rekenkundige en logische bewerkingen uitgevoerd op deze data. We hebben het getest, en afhankelijk daarvan verschillende delen van het programma uitgevoerd. In het bijzonder hebben we "geautomatiseerde" instructies zoals DJNZ gebruikt om programma's korter te maken. Andere instructies van deze laatste soort, LDDR, CPIR, INIR, worden in de rest van dit boek geintroduceerd.

Om programma's eenvoudiger te maken hebben we gebruik gemaakt van speciale eigenschappen van de Z80. In de meeste gevallen zijn deze programma's dan ook niet bruikbaar op een 8080.

We hebben kennis gemaakt met een structuur, die loop heet. Een andere belangrijke structuur komt nu aan bod: de subroutine.

SUBROUTINES

In principe is een subroutine een blok instructies, die door de programmeur een naam is gegeven. Praktisch gezien, moet de subroutine

beginnen met een speciale instructie: de *subroutine declaratie*, welke de subroutine herkenbaar maakt voor de vertaler. De subroutine moet ook afgesloten worden met een speciale instructie: de *return*. We zullen eerst het gebruik van een subroutine demonstreren, om de waarde ervan te laten zien. Daarna zullen we onderzoeken hoe we subroutines kunnen maken.

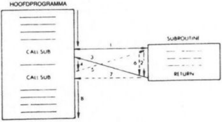


Fig. 3.35: Subroutine aanroep

In figuur 3.35 is het gebruik van een subroutine geillustreerd. Het hoofdprogramma staat links. De subroutine is symbolisch rechts weergegeven. We gaan nu het mechanisme onderzoeken. Het hoofdprogramma wordt normaal doorlopen totdat we de instructie "CALL SUB" tegenkomen. Dit is een *subroutine aanroep* en het gevolg is een sprong naar de subroutine. De eerste uit te voeren instructie na CALL SUB is de eerste instructie in de subroutine. Dit geeft pijl 1 aan in de figuur.

Dan wordt de subroutine uitgevoerd als ieder ander programma. We nemen hier aan dat de subroutine geen andere subroutine aanroepen bevat. De laatste instructie van de subroutine is een RETURN. D.m.v. deze instructie keren we terug naar het hoofdprogramma. De eerste nu uit te voeren instructie is de eerstvolgende instructie na CALL SUB in het hoofdprogramma. Dit is pijl 3 in de tekening. Het programma wordt daarna volgens pijl 4 doorlopen.

Opnieuw komen we in het hoofdprogramma een CALL SUB tegen. Ook nu springen we naar de subroutine (pijl 5). De subroutine wordt nogmaals doorlopen.

Komen we RETURN tegen, keren we terug naar het hoofdprogramma, naar de instructie die volgt op de laatste CALL SUB. Pijl 7 in de tekening. Het programma gaat door volgens pijl 8.

Het effect van de twee speciale instructies CALL SUB en RETURN

moet nu duidelijk zijn. Wat is de waarde van dit subroutine mechanisme?

De waarde ervan is, dat we een bepaald stuk programma overal kunnen gebruiken, zonder dat we het iedere keer opnieuw zouden moeten schrijven. Het eerste voordeel is, dat we geheugen uitsparen. Een tweede voordeel is, dat de programmeur slechts eenmaal een subroutine hoeft te schrijven, en deze toch overal in het programma kan gebruiken. Een vereenvoudiging dus bij het ontwerpen van programma's.

Oefening 3.31: Wat is het grootste nadeel van de subroutine? (Antwoord volgt.)

Het nadeel van d subroutine komt duidelijk naar voren, als we het verloop van het programa volgen. Het programma wordt langzamer, want er moeten enkele instructies meer worden uitgevoerd: CALL SUB en RETURN.

Uitvoering van het subroutine mechanisme

We gaan hier onderzoeken hoe de twee instructies CALL SUB en RETURN intern in de processor worden uitgevoerd. Het effect van CALL SUB is, dat een nieuwe instructie van een nieuw adres wordt gehaald. Je zult je herinneren (en anders lees je hoofdstuk 1 nog maar een keer), dat de programma teller PC dit adres bevat. Dat betekent, dat CALL SUB het adres van de subroutine moet laden in de programma teller. *Maar is dat alles wat deze instructie moet doen?*

Om die vraag te beantwoorden gaan we RETURN bekijken. Door deze instructie gaat het programma verder met de instructie die volgt op CALL SUB. Dat is alleen mogelijk als het adres van deze instructie ergens is bewaard. Dat adres is de waarde van de programma teller op het moment dat we CALLSUB tegen komen. PC wordt immers iedere keer als hij wordt gebruikt opgehoogd (zie ook hiervoor hoofdstuk 1). Hij bevat dan precies het adres dat we willen bewaren, zodat we later de RETURN uit kunnen voeren.

De volgende vraag is dan: waar bewaren we dat adres? Dat moet gebeuren op een plaats waar het niet gewist kan worden.

Bij dit alles moeten we de volgende situatie in acht nemen, geillustreerd door figuur 3.36. In dit voorbeeld bevat subroutine 1 een aanroep voor SUB 2. Ons mechanisme moet ook in deze situatie werken. Er kunnen zelfs meer dan twee subroutines voorkomen, zeg N "genes-

te" aanroepen. Na iedere CALL moet de waarde van PC opgeborgen worden. Dat betekent, dat er minstens 2N geheugen locaties voor dat doel moeten zijn. Daarbij komt dat we eerst moeten terugkeren uit SUB2 en dan pas uit SUB1. Met andere woorden, we hebben een structuur nodig, waarin de chronologische volgorde van de adressen wordt bewaard.

Deze structuur heeft een naam en die zijn we al eerder tegen gekomen: de *stapel*. Figuur 3.38 toont de inhoud van de stapel gedurende een aantal achtereenvolgende subroutine aanroepen. We kijken eerst naar het hoofdprogramma. Op adres 100 komen we de eerste aanroep tegen: CALL SUB1. Laten we aannemen dat de subroutine aanroep 3 bytes gebruikt (RST is daar een uitzondering op). Het volgende adres is dus niet "101", maar "103". De CALL instructie gebruikt de adressen "100", "101" en "102". De waarde van de programma teller is dus "103", want de processor weet dat de instructie CALL 3 bytes lang is. Aangezien "280" het adres van SUB1 is, is dat de waarde die geladen moet worden in PC.

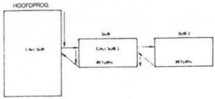


Fig. 3.36: Geneste aanroepen

Nu kunnen we het effect van de RETURN instructie en de juiste werking van ons stapel mechanisme laten zien. Het programma gaat door met de uitvoering van SUB2, totdat het RETURN tegenkomt op tijdstip tijd 3. Door de RETURN wordt d.m.v. een POP de top van de stapel in de programma teller geplaatst. M.a.w. PC krijgt de zelfde waarde als vlak voor de subroutine aanroep. De top van de stapel is in ons voorbeeld "303". Figuur 3.38 laat zien, dat op tijdstip tijd 3 "303" van de stapel wordt verwijderd en in de programma teller wordt gestopt. Op tijd 4 komt het programma de RETURN van SUB1 tegen. Top van de stapel is "103". Ook deze waarde verhuist naar PC. De processor gaat dus door met het hoofdprogramma vanaf adres "103". Dat is precies wat we willen hebben. Figuur 3.38 laat zien, dat op tijd 4 de stapel leeg is. Het mechanisme werkt.

Dit mechanisme werkt zolang de stapel niet vol is. Daarom hadden de eerste microprocessors met een vier- of acht-register stapel een beperkt aantal (4 of 8) subroutine niveaus.

In de figuren 3.36 en 3.37 zijn de subroutines rechts van het hoofdprogramma getekend. Dat is alleen maar voor de duidelijkheid gedaan. In de praktijk zijn subroutines normale instructies in het programma. Ze kunnen overal in het programma staan, in het begin, in het midden of aan het eind. Daarom worden ze voorafgegeaan door een subroutine declaratie: ze moeten herkenbaar zijn. Deze instructies vertellen de vertaler, dat dat wat volgt een subroutine is. Dit soort aanwijzingen voor de vertaler komen in hoofdstuk 10 aan de orde.

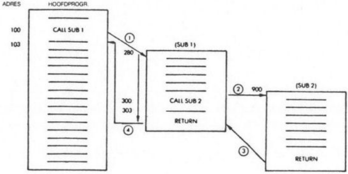


Fig. 3.37: De subroutine aanroepen

STACK:	Tijdstip (1)	Tijdstip (2)	Tijdstip (3)	Tijdstip (4)
	103	103	103	
		303		

Fig. 3.38: De stapel in de tijd

Z80 subroutines

De basis beginselen m.b.t. subroutines zijn nu behandeld. Om dit mechanisme uit te kunnen voeren is een stapel nodig. De Z80 beschikt over een 16-bits stack-pointer register. De stapel kan zich daarom overal in het geheugen bevinden, en mag tot 64K (1K = 1024) bytes groot zijn, aangenomen dat deze hoeveelheid ook daadwerkelijk beschikbaar is. Het start adres en de maximale afmetingen van de stapel worden door de programmeur bepaald voor deze zijn programma schrijft. Een gedeelte van het geheugen wordt dan gereserveerd voor de stapel.

De Z80 heeft twee soorten subroutine aanroepen. De directe of onvoorwaardelijke aanroep hebben we al leren kennen: CALL ADRES. De Z80 beschikt ook over een voorwaardelijke aanroep, waarmee naar een subroutine wordt gesprongen als aan een bepaalde voorwaarde is voldaan. Een voorbeeld hiervan is: CALL NZ,SUB1, waardoor naar SUB1 wordt gesprongen, als het resultaat van de vorige instructie niet nul is. Het is een krachtige instructie, aangezien vele subroutine aanroepen voorwaardelijk zijn.

CALL CC,NN wordt alleen uitgevoerd, als aan de conditie gespecificeerd door CC is voldaan. CC bestaat uit drie bits (bit 4, 5 en 6 van de opcode), waarmee acht voorwaarden kunnen worden gespecificeerd. Ze hebben allemaal betrekking op de statusbits "Z", "C", "P/V" en "S", al dan niet gelijk aan nul.

Op gelijke wijze bezit de Z80 twee soorten terugkeer instructies: RET en RET CC.

RET is de basis terugkeer instructie, en bestaat uit 1 byte. De twee bytes op de top van de stapel worden in PC geladen. De instructie is onvoorwaardelijk.

RET CC doet precies het zelfde, op voorwaarde dat de conditie gespecificeerd door CC waar is. De CC bits hebben de zelfde betekenis als bij de CALL.

Om interrupt subroutines af te sluiten heeft de Z80 nog twee andere instructies: RETI en RETN. Ze worden zowel bij de behandeling van de interrupts als bij de instructies besproken.

Tenslotte heeft de Z80 een speciale instructie, die lijkt op een subroutine aanroep, maar waarmee alleen gesprongen kan worden naar een van acht start adressen op pagina 0. Het is de RST P instructie. Deze een-byte grote instructie bewaart PC op de stapel en springt naar een adres, gespecificeerd door het P veld. Bits 4, 5 en 6 van de instructie bevatten het P veld, dat met acht wordt vermenigvuldigd.

Dus als P "000" is, wordt gesprongen naar "00H"; als P "001" is naar "08H" enz. P is maximaal "111", wat correspondeert met adres "38H". Deze instructie is erg snel, want het heeft maar een byte. Hij kan echter maar springen naar acht adressen op pagina 0. Deze adressen zijn verder slechts acht bytes van elkaar verwijderd. De instructie is overgenomen van de 8080, en wordt vaak bij interrups gebruikt. Hij wordt beschreven in het interrupt hoofdstuk. De programmeur kan de instructie echter ook voor andere doeleinden gebruiken, omdat het in feite een speciale subroutine aanroep is.

Subroutine voorbeelden

De meeste programma's die we hebben gemaakt, en die we nog zullen maken zullen gewoonlijk zijn geschreven als subroutines. Het vermenigvuldig programma bijvoorbeeld, zal waarschijnlijk op veel plaatsen in een programma gebruikt worden. Om de ontwikkeling van dat programma gemakkelijker en doorzichtiger te maken, maken we van het vermenigvuldig programma een subroutine en geven het een toepasselijke naam: VERM. Aan het eind van de routine plaatsen we de instructie RET.

Oefening 3.32: Als VERM wordt gebruikt als subroutine, "beschadigt" het dan interne status bits of registers?

Recursie

Als een subroutine zichzelf aanroept, dan heet die subroutine recursief. Als je het subroutine mechanisme hebt begrepen, zul je de volgende vraag kunnen beantwoorden:

Oefening 3.33: Mag een subroutine zichzelf aanroepen? (M.a.w blijft alles goed werken?) Als je niet zeker bent, teken dan de stapel en vul deze in met de opeenvolgende adressen. Bekijk dan de registers en het geheugen (zie oefening 3.18) en onderzoek deze op eventuele problemen.

Interrupts worden nader bekeken in het input/output hoofdstuk (hoofdstuk 6). Alle returns zijn 1 byte, en alle calls 3 bytes lang, behalve RST.

Oefening 3.34: Waarom is de terugkeer van een subroutine sneller dan de aanroep ervan? (Tijden zijn te vinden in het volgende hoofdstuk.). (Hint: bekijk opnieuw het subroutine mechanisme en analiseer de interne bewerkingen die uitgevoerd moeten worden, als het antwoord niet meteen duidelijk is.)

Subroutine parameters

Normaal is, dat een subroutine een bewerking uitvoert op data. De vermenigvuldig subroutine, bijvoorbeeld, moet weten hoe groot de deler en het deeltal zijn. Deze subroutine verwachtte deze getallen in het geheugen te vinden. Dit illustreert een manier om de parameters door te geven aan de subroutine: via het geheugen. Er zijn nog twee andere manieren. In totaal zijn er dus drie manieren:

1 - via de registers
2 - via het geheugen
3 - via de stapel

Registers kunnen worden gebruikt om parameters door te geven. Als de registers beschikbaar zijn is dat een voordelige oplossing, want de subroutine blijft op deze wijze onafhankelijk van het geheugen. Wordt een vaste geheugen locatie gebruikt, dan zullen de andere gebruikers zeer oplettend moeten zijn, dat ze de zelfde afspraken hanteren. Bovendien moeten ze zich ervan vergewissen dat die geheugen plaatsen daadwerkelijk beschikbaar zijn. Daarom wordt vaak alleen voor dit doel een bepaald gedeelte van het geheugen gereserveerd.

Geheugen heeft het voordeel dat het meer data kan bevatten, de snelheid van het programma neemt echter af. Bovendien is de subroutine dan gebonden aan een bepaald gedeelte van het geheugen.

De stapel heeft het zelfde voordeel als de registers: het gebruik ervan is geheugen onafhankelijk. De subroutine weet eenvoudig, dat de data op de stapel te vinden zijn. Natuurlijk heeft ook deze methode zijn nadelen. Omdat data op de stapel komen te staan, vermindert daarmee het mogelijke aantal subroutine niveaus. Het gebruik van de stapel wordt ook gecompliceerder, waardoor misschien meerdere stapels nodig worden.

De keus is aan de programmeur. Maar in het algemeen willen we zo lang als dat mogelijk is onafhankelijk van het geheugen blijven.

De stapel is een mogelijke oplossing als er geen registers beschikbaar zijn. De overdracht van grote hoeveelheden data is echter vaak alleen via het geheugen mogelijk. Een elegante oplossing daarvoor is een verwijsadres of pointer, die naar een blok data wijst, door te geven aan de

subroutine. De pointer wijst naar het beginadres van het blok met data. Via een register of de stapel kan deze pointer aan de subroutine doorgegeven worden. We kunnen de pointer ook op een bepaalde plaats in het geheugen plaatsen.

Is geen van deze twee oplossingen mogelijk, dan moeten we met de routine afspreken, dat deze de data op een bepaalde plaats in het geheugen kan vinden.

Oefening 3.35: Welke is de beste methode voor recursief gebruik?

Subroutine bibliotheek

Het heeft een duidelijk voordeel als van delen van het programma gestructureerde subroutines worden gemaakt: ze kunnen onafhankelijk van elkaar op punt gesteld worden en hun doel is duidelijk door de naam die we ze geven. Subroutines kunnen ook gebruikt worden in andere delen van het programma. We kunnen een bibliotheek opbouwen van bruikbare subroutines. Het gebruik van routines uit een dergelijke bibliotheek kan ook nadelen hebben: het kan zijn dat de subroutine meer doet dan we eigenlijk willen, of dat er andere registers worden gebruikt dan we willen. Het gebruik van de bibliotheek kan dus ten kosten van de efficientie gaan. De oplettende programmeur zal de nadelen tegen de voordelen afwegen.

SAMENVATTING

In dit hoofdstuk hebben we bekeken op welke wijze data in de Z80 door de instructies wordt gemanipuleerd. In moeilijkheid toenemende algoritmes zijn geintroduceerd en vertaald in programma's. De belangrijkste soorten instructies zijn behandeld en gebruikt.

Belangrijke structuren als loops, stapels en subroutines zijn gedefinieerd.

Je moet nu enig begrip hebben gekregen van het programmeren en de in standaard toepassingen gebruikte technieken. In het volgende hoofdstuk worden de beschikbare instructies behandeld.

```
         A=00 BC=0000 DE=0000 HL=0000 S=0300 P=0100 0100' LD   BC,(0200)
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00        (0200')
         A=00 BC=0003 DE=0000 HL=0000 S=0300 P=0104 0104' LD   B,08
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
         A=00 BC=0803 DE=0000 HL=0000 S=0300 P=0106 0106' LD   DE,(0202)
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00        (0202')
         A=00 BC=0803 DE=0005 HL=0000 S=0300 P=010A 010A' LD   D,00
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
         A=00 BC=0803 DE=0005 HL=0000 S=0300 P=010C 010C' LD   HL,0000
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00        (0000')
         A=00 BC=0803 DE=0005 HL=0000 S=0300 P=010F 010F' SRL  C
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
      C  A=00 BC=0801 DE=0005 HL=0000 S=0300 P=0111 0111' JR   NC,0114
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00        (0114')
      C  A=00 BC=0801 DE=0005 HL=0000 S=0300 P=0113 0113' ADD  HL,DE
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
         A=00 BC=0801 DE=0005 HL=0005 S=0300 P=0114 0114' SLA  E
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
  V      A=00 BC=0801 DE=000A HL=0005 S=0300 P=0116 0116' RL   D
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V      A=00 BC=0801 DE=000A HL=0005 S=0300 P=0118 0118' DEC  B
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
    N    A=00 BC=0701 DE=000A HL=0005 S=0300 P=0119 0119' JP   NZ,010F
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00        (010F')
    N    A=00 BC=0701 DE=000A HL=0005 S=0300 P=010F 010F' SRL  C
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V C    A=00 BC=0700 DE=000A HL=0005 S=0300 P=0111 0111' JR   NC,0114
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00        (0114')
Z V C    A=00 BC=0700 DE=000A HL=0005 S=0300 P=0113 0113' ADD  HL,DE
        A'=00 B'=0000 D'=0000. H'=0000 X=0000 Y=0000 I=00
Z V      A=00 BC=0700 DE=000A HL=000F S=0300 P=0114 0114' SLA  E
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
  V      A=00 BC=0700 DE=0014 HL=000F S=0300 P=0116 0116' RL   D
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V      A=00 BC=0700 DE=0014 HL=000F S=0300 P=0118 0118' DEC  B
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
    N    A=00 BC=0600 DE=0014 HL=000F S=0300 P=0119 0119' JP   NZ,010F
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00        (010F')
    N    A=00 BC=0600 DE=0014 HL=000F S=0300 P=010F 010F' SRL  C
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V      A=00 BC=0600 DE=0014 HL=000F S=0300 P=0111 0111' JR   NC,0114
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00        '0114')
Z V      A=00 BC=0600 DE=0014 HL=000F S=0300 P=0114 0114' SLA  E
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
  V      A=00 BC=0600 DE=0028 HL=000F S=0300 P=0116 0116' RL   D
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V      A=00 BC=0600 DE=0028 HL=000F S=0300 P=0118 0118' DEC  B
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
    N    A=00 BC=0500 DE=0028 HL=000F S=0300 P=0119 0119' JP   NZ,010F
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00        (010F')
    N    A=00 BC=0500 DE=0028 HL=000F S=0300 P=010F 010F' SRL  C
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V      A=00 BC=0500 DE=0028 HL=000F S=0300 P=0111 0111' JR   NC,0114
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00        (0114')
Z V      A=00 BC=0500 DE=0028 HL=000F S=0300 P=0114 0114' SLA  E
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
  V      A=00 BC=0500 DE=0050 HL=000F S=0300 P=0116 0116' RL   D
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V      A=00 BC=0500 DE=0050 HL=000F S=0300 P=0118 0118' DEC  B
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
    N    A=00 BC=0400 DE=0050 HL=000F S=0300 P=0119 0119' JP   NZ,010F
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00        (010F')
    N    A=00 BC=0400 DE=0050 HL=000F S=0300 P=010F 010F' SRL  C
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
```

Fig. 3.39: Vermenigvuldiging: het programma volledig doorlopen

```
Z V      A=00 BC=0400 DE=0050 HL=000F S=0300 P=0111 0111' JR   NC,0114
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00        (0114')
Z V      A=00 BC=0400 DE=0050 HL=000F S=0300 P=0114 0114' SLA  E
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
S  V     A=00 BC=0400 DE=00A0 HL=000F S=0300 P=0116 0116' RL   D
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V      A=00 BC=0400 DE=00A0 HL=000F S=0300 P=0118 0118' DEC  B
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
    N    A=00 BC=0300 DE=00A0 HL=000F S=0300 P=0119 0119' JP   NZ,010F
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00        (010F')
    N    A=00 BC=0300 DE=00A0 HL=000F S=0300 P=010F 010F' SRL  C
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V      A=00 BC=0300 DE=00A0 HL=000F S=0300 P=0111 0111' JR   NC,0114
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00        (0114')
Z V      A=00 BC=0300 DE=00A0 HL=000F S=0300 P=0114 0114' SLA  E
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
      C  A=00 BC=0300 DE=0040 HL=000F S=0300 P=0116 0116' RL   D
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
         A=00 BC=0300 DE=0140 HL=000F S=0300 P=0118 0118' DEC  B
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
    N    A=00 BC=0200 DE=0140 HL=000F S=0300 P=0119 0119' JP   NZ,010F
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00        (010F')
    N    A=00 BC=0200 DE=0140 HL=000F S=0300 P=010F 010F' SRL  C
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V      A=00 BC=0200 DE=0140 HL=000F S=0300 P=0111 0111' JR   NC,0114
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00        (0114')
Z V      A=00 BC=0200 DE=0140 HL=000F S=0300 P=0114 0114' SLA  E
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
S        A=00 BC=0200 DE=0180 HL=000F S=0300 P=0116 0116' RL   D
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
         A=00 BC=0200 DE=0280 HL=000F S=0300 P=0118 0118' DEC  B
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
    N    A=00 BC=0100 DE=0280 HL=000F S=0300 P=0119 0119' JP   NZ,010F
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00        (010F')
    N    A=00 BC=0100 DE=0280 HL=000F S=0300 P=010F 010F' SRL  C
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V      A=00 BC=0100 DE=0280 HL=000F S=0300 P=0111 0111' JR   NC,0114
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00        (0114')
Z V      A=00 BC=0100 DE=0280 HL=000F S=0300 P=0114 0114' SLA  E
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V C    A=00 BC=0100 DE=0200 HL=000F S=0300 P=0116 0116' RL   D
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
  V      A=00 BC=0100 DE=0500 HL=000F S=0300 P=0118 0118' DEC  B
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z   N    A=00 BC=0000 DE=0500 HL=000F S=0300 P=0119 0119' JP   NZ,010F
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00        (010F')
Z   N    A=00 BC=0000 DE=0500 HL=000F S=0300 P=011C 011C' LD   (0204),HL
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00        (0204')
Z   N    A=00 BC=0000 DE=0500 HL=000F S=0300 P=011F 011F' NOP
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
```

Fig. 3.39: Vermenigvuldiging: het programma volledig doorlopen (vervolg)

ANTWOORD OP OEFENING 3.18 (VERMENIGVULDIGING)

```
CROMEMCO CDOS Z80 ASSEMBLER version 02.15                 PAGE 0001

0000'           0001        ORG    0100H
    (0200)      0002 MPRAD  EQU    0200H
    (0202)      0003 MPDAD  EQU    0202H
    (0204)      0004 RESAD  EQU    0204H
                0005 ;
0100  ED4B0002  0006 MP488  LD     BC,(MPRAD)      ;LOAD MULTIPLIER INTO C
0104  0608      0007        LD     B,8             ;B IS BIT COUNTER
0106  ED5B0202  0008        LD     DE,(MPDAD)      ;LOAD MULTIPLICAND INTO E
010A  1600      0009        LD     D,0             ;CLEAR D
010C  210000    0010        LD     HL,0            ;SET RESULT TO 0
010F  CB39      0011 MULT   SRL    C               ;SHIFT MULTIPLIER BIT INTO CARRY
0111  3001      0012        JR     NC,NOADD        ;TEST CARRY
0113  19        0013        ADD    HL,DE           ;ADD MPD TO RESULT
0114  CB23      0014 NOADD  SLA    E               ;SHIFT MPD LEFT
0116  CB12      0015        RL     D               ;SAVE BIT IN D
0118  05        0016        DEC    B               ;DECREMENT SHIFT COUNTER
0119  C20F01    0017        JP     NZ,MULT         ;DO IT AGAIN IF COUNTER <> 0
011C  220402    0018        LD     (RESAD),HL      ;STORE RESULT
011F  (0000)    0019        END

Errors          0
```

Fig. 3.40: Het vermenigvuldig programma (hex)

LABEL	INSTRUCTIE	B	C	C (CARRY)	D	E	H	L
		00	00	0	00	00	00	00
MP488	LD BC,(0200)	00	03	0	00	00	00	00
	LD B,08	08	03	0	00	00	00	00
	LD DE,(0202)	08	03	0	00	05	00	00
	LD D,00	08	03	0	00	05	00	00
	LD HL,0000	08	03	0	00	05	00	00
MULT	SRL C	08	01	1	00	05	00	00
	JR NC,0114	08	01	1	00	05	00	00
	ADD HL,DE	08	01	1	00	05	00	05
NOADD	SLA E	08	01	0	00	0A	00	05
	RL D	08	01	0	00	0A	00	05
	DEC B	07	01	0	00	0A	00	05
	JP NZ,010F	07	01	0	00	0A	00	05
MULT	SRL C	07	00	1	00	0A	00	05
	JR NC,0114	07	00	1	00	0A	00	05
	ADD HL,DE	07	00	0	00	0A	00	0F
NO ADD	SLA E	07	00	0	00	14	00	0F
	RL D	07	00	0	00	14	00	0F
	DEC B	06	00	0	00	14	00	0F
	JP NZ,010F	06	00	0	00	14	00	0F

Fig. 3.41: De loop twee maal doorlopen

4

DE Z80 INSTRUCTIE SET

INLEIDING

In dit hoofdstuk worden eerst de verschillende klassen van instructies die in een general-purpose computer beschikbaar zijn besproken. Daarna komen een voor een de instructies van de Z80 aan bod. Daarbij wordt tot in detail ingegaan op hun doel en op de wijze waarop ze de vlaggen beinvloeden. Ook hun gebruik met de verschillende adresseer methoden komt aan de orde. In hoofdstuk 5 worden deze adresseer technieken uitgebreid besproken.

INSTRUCTIE KLASSEN

Instructies kunnen op vele manieren in klassen worden verdeeld, en er is geen standaard. Wij zullen de volgende indeling maken:

1 - data overdrachten
2 - data bewerkingen
3 - testen en sprongen
4 - input/output
5 - controle of besturing

We gaan deze klassen stuk voor stuk nader beschouwen.

Data overdrachten

Deze instructies verplaatsen data tussen registers, of tussen een register en geheugen, of tussen een register en een input/output apparaat. Er kunnen speciale instructies voorkomen voor bepaalde belangrijke registers. Push en pop instructies zijn er, bijvoorbeeld, om de stapel efficient te kunnen gebruiken. Ze transporteren data tussen stapel en de accumulator in 1 instructie, terwijl tegelijk het stack-pointer register de nieuwe goede waarde krijgt.

Data bewerkingen

Deze klasse valt uiteen in de volgende categorieen:

1 - rekenkundige bewerkingen (zoals plus en minus)
2 - bit manipulatie (set en reset)
3 - "increment" en "decrement" (resp. 1 optellen en 1 aftrekken)
4 - logische bewerkingen (AND, OR en XOR)
5 - verschuivingen (verschuiven en roteren)

Hierbij moet worden opgemerkt, dat om efficient data te kunnen verwerken krachtige rekenkundige instructies nodig zijn. Zoals de vermenigvuldiging en de deling. Helaas zijn ze meestal niet beschikbaar op de meeste microprocessors. Ook is het wenselijk te beschikken over goede schuif instructies, of over instructies waarmee nibbles verwisseld kunnen worden. Deze zijn echter eveneens vaak afwezig bij microprocessors.

Voordat we de werkelijke Z80 instructies bekijken, trachten we nog even het verschil tussen *verschuiven* en *roteren* in herinnering te brengen. De verschuif instructie schuift de inhoud van een register een bit naar links of naar rechts. Het bit dat uit het register valt, komt in het carry bit terecht. Het binnenschuivende bit heeft de waarde 0, behalve bij een rekenkundige verschuiving naar rechts, waarbij het MSB wordt gedupliceerd.

Bij de rotatie komt het uitschuivende bit ook in de carry terecht, maar het naar binnen schuivende bit krijgt de vorige waarde van het carry bit. Dit is in feite een rotatie van 9 bits. Vaak wordt een 8-bits rotatie gewenst. Het naar binnen schuivende bit heeft dan de zelfde waarde als het naar buiten schuivende bit. Buiten de Z80 bezitten slechts weinig processors deze instructie (Zie figuur 4.1).

Tenslotte is het gemakkelijk de beschikking te hebben over nog een

andere soort verschuiving: de rekenkundige verschuiving naar rechts. Bij het werken met twee-complement getallen, in het bijzonder met floating point getallen, moeten vaak negatieve getallen naar rechts worden verschoven. Daarbij moet het teken bewaard blijven. Het binnenschuivende bit moet een 1 zijn bij negatieve getallen. Het naar binnen schuivende bit moet gelijk zijn aan zijn voorganger. Het teken wordt als het ware verlengd. Dat doet de rekenkundige verschuiving naar rechts.

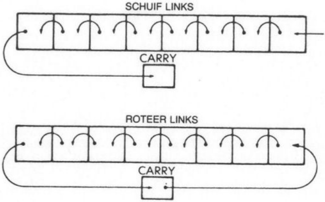


Fig. 4.1: Verschuiving en rotatie

Testen en sprongen

De test instructies testen bits van registers op 0 of 1 of een combinatie daarvan. Minimaal moeten de vlaggen getest kunnen worden. Daarom moeten er zoveel mogelijk vlaggen in het vlaggen of status register worden gestopt. Bovendien is het gemakkelijk combinaties van vlaggen of andere bits te kunnen testen m.b.v. een instructie. Tenslotte is het wenselijk dat ieder bit van ieder register te testen is, of de waarde van een register te testen ten opzichte van de waarde van een ander register (groter dan, kleiner dan, of gelijk aan). Microprocessors kunnen meestal alleen de bits van het status register testen. Ook hier biedt de Z80 meer mogelijkheden dan de meeste andere processors.

De sprong instructies zijn onder te verdelen in drie categorieen:

1 - sprong met een volledig 16-bits adres
2 - de relatieve sprong, welke een vaak 8-bits grote verplaatsing specificeert
3 - de call, gebruikt bij subroutines.

Het is gemakkelijk een instructie te hebben waarmee twee of zelfs drie richtingen uit gesprongen kan worden (denk bijvoorbeeld aan een vergelijking van twee getallen: groter dan, gelijk aan, of kleiner dan, waarbij afhankelijk van de uitkomst van de vergelijking naar een bepaald adres wordt gesprongen.). Het is ook gemakkelijk te beschikken over een instructie, waarmee slechts een paar instructies voor of achteruit gesprongen kan worden. Tenslotte willen we graag een test-en-spring instructie, die zelf een teller bijhoudt en die teller test. Erg gemakkelijk bij loops. Helaas niet beschikbaar in de meeste microprocessors. De Z80 heeft wel een dergelijke instructie, hoewel deze alleen met register B werkt.

Input/output

Input/output instructies verzorgen de in- en uitvoer m.b.v input/output apparatuur. De meerderheid van de processors maakt gebruik van *memory-mapped I/O*: input/output apparatuur wordt aan de adres bus gekoppeld, alsof het normaal geheugen is. De apparatuur wordt ook als zodanig geadresseerd. Dit soort instructies zijn meestal drie of meer bytes lang, en dus langzaam. Om in zo'n geval toch snel I/O te kunnen plegen, is het gewenst dat het systeem beschikt over een verkorte adresserings mogelijkheid. Vaak is dat mogelijk met adressen op pagina 0. Maar ook dan is een efficient gebruik meestal niet mogelijk, daar pagina 0 gebruikt wordt door RAM. De Z80 heeft, net als de 8080, wel speciale I/O instructies. De ontwerper heeft dus de keus: I/O via het geheugen, of m.b.v. I/O instructies.
De instructies komen later dit hoofdstuk nog aan de orde.

Controle of besturings instructies

Controle instructies voorzien het systeem van synchronisatie signalen en kunnen een programma staken of tijdig onderbreken. (Dit wordt in hoofdstuk 6 besproken.)

DE Z80 INSTRUCTIE SET

Inleiding

De Z80 is ontworpen als vervanger, met verbeteringen, van de 8080. Het resultaat van deze filosofie is, dat de Z80 alle instructies van de 8080 kent, plus nog enkele meer. Als je het beperkte aantal bits in een 8-bits opcode in aanmerking neemt, mag je je afvragen hoe de ontwerpers van de Z80 dat allemaal voor elkaar hebben gekregen. Ze hebben dat gedaan door de niet gebruikte 8080 opcodes te gebruiken, en door een byte toe te voegen aan geindexeerde instructies. Daarom nemen sommige Z80 instructies tot 4 bytes geheugen ruimte in beslag.

Het is belangrijk te onthouden, dat een programma op veel manieren geschreven kan worden. Om goed te kunnen programmeren is een grondige kennis van de instructie set vereist. Tijdens het leren programmeren hoeven we echter geen optimale programma's te maken. Daarom is het bij de eerste keer doornemen van dit hoofdstuk niet belangrijk dat je alle instructies uit het hoofd kent. Wel is belangrijk, dat je alle categorieen kent, en dat je de voorbeelden goed bestudeert. Daarna, tijdens het schrijven van programma's, kun je de instructie set eens goed doornemen, en de meest geschikte instructies kiezen voor het programma. Deze paragraaf is bedoeld om de instructies duidelijk te maken, en om ze in de diverse categorieen te plaatsen. De lezer die geinteresseerd is in de preciese werking van de instructies, kan aan het eind van dit hoofdstuk terecht: daar wordt iedere instructie apart behandeld.

Nu gaan we door met ons onderzoek naar de mogelijkheden van de vijf categorieen instructies, zoals we die aan het begin van dit hoofdstuk hebben gedefinieerd.

Data overdracht instructies

Deze instructies kunnen weer in vier categorieen worden onderverdeeld: 8-bits overdrachten, 16-bits overdrachten, bewerkingen van de stapel, en blok overdrachten.

Acht-bits data overdrachten

De "load" instructies voeren deze overdrachten uit. het formaat van deze instructies is:

LD bestemming,bron

Bijvoorbeeld: de inhoud van register B moet in de accumulator geladen worden:

LD B,A

We kunnen dus direct data overdragen van een van de werkregisters naar een ander. (Registers ABCDEHL)

Om een werkregister (behalve de accumulator) te laden vanuit een geheugen locatie, moet dat adres eerst in een register paar worden geladen, bijvoorbeeld H en L.

Om, bijvoorbeeld, C te laden met de inhoud van adres "1234" (we gebruiken een 16-bits "load" instructie), laden we eerst H en L met 1234. Met de instructie

LD C,(HL)

wordt dan het gewenste resultaat bereikt.

Een uitzondering is de accumulator. Dit register kan direct vanuit ieder geheugen adres geladen worden. Dat heet de verlengde adresserings mode.

LD A,(1234)

laadt de accumulator met de inhoud vañ geheugen adres 1234.
NB: (...) betekent: de inhoud van ...

De instructie staat op de volgende manier in het geheugen:

adres	PC :3A	(opcode)
	PC + 1:34	(lage orde deel adres)
	PC + 2:12	(hoge orde deel van het adres)

Het adres wordt dus in omgekeerde volgorde opgeslagen:

3A	adres laag	adres hoog

Alle werkregisters kunnen geladen worden met een 8-bits waarde of "literal", welke het tweede byte van de instructie vormt. Dit heet directe adressering. Een voorbeeld is:

LD E,12H

waardoor register E geladen wordt met de waarde 12 hexadecimaal.
Geheugen:

PC: 1E (opcode)
PC + 1: 12 (literal)

		SOURCE															
		IMPLIED		REGISTER							REG INDIRECT			INDEXED		EXT. ADDR.	IMME.
		I	R	A	B	C	D	E	H	L	(HL)	(BC)	(DE)	(IX + d)	(IY + d)	(nn)	n
DESTINATION REGISTER	A	ED 57	ED 5F	7F	78	79	7A	7B	7C	7D	7E	0A	1A	DD 7E d	FD 7E d	3A n n	3E n
	B			47	40	41	42	43	44	45	46			DD 46 d	FD 46 d		06 n
	C			4F	48	49	4A	4B	4C	4D	4E			DD 4E d	FD 4E d		0E n
	D			57	50	51	52	53	54	55	56			DD 56 d	FD 56 d		16 n
	E			5F	58	59	5A	5B	5C	5D	5E			DD 5E d	FD 5E d		1E n
	H			67	60	61	62	63	64	65	66			DD 66 d	FD 66 d		26 n
	L			6F	68	69	6A	6B	6C	6D	6E			DD 6E d	FD 6E d		2E n
REG INDIRECT	(HL)			77	70	71	72	73	74	75							36 n
	(BC)			02													
	(DE)			12													
INDEXED	(IX+d)			DD 77 d	DD 70 d	DD 71 d	DD 72 d	DD 73 d	DD 74 d	DD 75 d							DD 36 d n
	(IY+d)			FD 77 d	FD 70 d	FD 71 d	FD 72 d	FD 73 d	FD 74 d	FD 75 d							FD 36 d n
EXT. ADDR	(nn)			32 n n													
IMPLIED	I			ED 47													
	R			ED 4F													

Fig. 4.2: 8-bits LOAD groep-"LD"

De geindexeerde adressering is ook beschikbaar voor het laden van registers, en zal volledig worden beschreven in het volgende hoofdstuk, dat gaat over adresseer technieken. Om specifieke registers te laden bestaan nog andere instructies. Een tabel met daarin alle mogelijkheden staat in figuur 4.2. Deze tabellen zijn door ZILOG INC. beschikbaar gesteld. De grijze hokjes geven de instructies die de Z80 gemeen heeft met de 8080.

16-bits data overdrachten

In principe kan elk van de 16-bits register paren BC, DE, HL, SP, IX, IY geladen worden met een 16-bits literal, of van de top van de stapel, of van een bepaald geheugen adres. De instructie kan ook worden omgekeerd: Het register kan zijn inhoud opbergen in het geheugen, of op de top van de stapel (in feite ook in het geheugen, en wel op het adres (IP)). Bovendien kan SP geladen worden vanuit de register paren HL, IX, en IY. Op deze wijze kunnen meerdere stapels gemaakt worden.

		BRON									
		REGISTER							IMM. EXT.	EXT. ADDR.	REG. INDIR.
		AF	BC	DE	HL	SP	IX	IY	nn	(nn)	(SP)
BESTEMMING REGISTER	AF										F1
	BC								01 n n	ED 4B n n	C1
	DE								11 n n	ED 5B n n	D1
	HL								21 n n	2A n n	E1
	SP				[illegible]	-	DD F9	FD F9	31 n n	ED 7B n n	
	IX								DD 21 n n	DD 2A n n	DD E1
	IY								FD 21 n n	FD 2A n n	FD E1
EXT. ADDR.	(nn)		ED 43 n n	ED 53 n n	[illegible]	ED 73 n n	DD 22 n n	FD 22 n n			
PUSH INSTRUCTIONS → REG. IND.	(SP)	[illegible]	[illegible]	[illegible]	[illegible]		DD E5	FD E5			

NOTE: The Push & Pop Instructions adjust the SP after every execution

↑ POP INSTRUCTIONS

Fig. 4.3: 16-bits LOAD groep-"LD", "PUSH" en "POP"

Ook hier staan alle mogelijkheden in een tabel. Push en pop instructies behoren ook tot deze categorie. Deze instructies regelen immers het verkeer tussen stapel en registers. Opgemerkt moet nog worden, dat er geen push en pop instructies zijn die slechts 8 bits tegelijk redden.

Een push of pop wordt altijd uitgevoerd met een register paar: AF, BC, DE, HL, IX, of IY. (Zie de onderste rij en de rechtse kolom van figuur 4.3).

Is het register paar een van de volgende: AF, BC, DE, of HL, dan is de instructie een byte lang. Efficient dus.
Neem bijvoorbeeld aan, dat SP de waarde "0100" heeft. De volgende instructie wordt uitgevoerd:

PUSH AF

Dan gebeurt het volgende: SP wordt eerst met 1 verminderd. De inhoud van A wordt op de top van de stapel geplaatst. SP wordt nog eens met 1 verminderd. De inhoud van F komt op de stapel. Aan het eind van de instructie wijst SP dus naar de top van de stapel, wat in ons voorbeeld de waarde van F is.

Belangrijk is te onthouden, dat bij de Z80 SP naar de top van de stapel wijst, en dat SP kleiner wordt als er een push plaats vindt. Andere processors hanteren vaak andere afspraken, wat veel verwarring kan veroorzaken.

		IMPLICIETE ADRESSERING				
		AF'	BC', DE' & HL'	HL	IX	IY
IMPLIED	AF	08				
	BC, DE & HL		D9			
	DE			EB		
REG. INDIR.	(SP)			E3	DD E3	FD E3

Fig. 4.4: Verwissel instructies "EX" en "EXX"

Verwissel instructies

De Z80 heeft nog een speciale data overdracht instructie: EX. Het voert een dubbele data overdracht uit. Het verwisselt (EX is een afkorting van exchange, wat verwisseling betekent) de inhouden van *twee* gespecificeerde adressen. De instructie kan gebruikt worden om de top van de stapel te verwisselen met HL, IX of IY. Of om de inhouden van DE en HL of AF en AF' te verwisselen (AF' is het andere AF register paar van de Z80).

Tenslotte is de instructie EXX beschikbaar om de inhouden van BC, DE en HL te verwisselen tegen de inhouden van de corresponderende registers in de tweede geheugen bank van de Z80.

Zie voor deze groep instructies figuur 4.4.

			BRON REG. INDIR. (HL)	
BESTEMMING	REG. INDIR.	(DE)	ED A0	'LDI' – Load (DE)←(HL) Inc HL & DE, Dec BC
			ED B0	'LDIR,' – Load (DE)←(HL) Inc HL & DE, Dec BC, Repeat until BC = 0
			ED A8	'LDD' – Load (DE)←(HL) Dec HL & DE, Dec BC
			ED B8	'LDDR' – Load (DE)←(HL) Dec HL & DE, Dec BC, Repeat until BC = 0

Reg HL points to source
Reg DE points to destination
Reg BC is byte counter

Fig. 4.5: block overdracht instructies-"LDI", "LDIR", "LOD" en "LDDR".

Blok overdracht instructies

M.b.v.deze instructies worden blokken data verplaatst i.p.v. een of twee bytes. Omdat deze instructies voor de fabrikant moeilijker in te bouwen zijn dan de meeste andere instructies komen we ze niet vaak tegen bij andere processors. Ze zijn gemakkelijk bij het programme-

ren en kunnen het programma vaak aanzienlijk beter maken, vooral bij in- en uitvoer. Hun gebruik en hun voordelen zullen het hele boek door worden gedemonstreerd. De Z80 heeft wel enkele automatische blok overdracht instructies. We moeten ons bij het gebruik daarvan houden aan enkele afspraken.

We hebben altijd drie register paren nodig: BC, DE en HL:

BC is een 16-bits teller. 2 tot de macht 16 bytes (dat is 64K) kunnen worden verplaatst. HL wijst de oorsprong aan. Dat mag overal in het geheugen zijn. DE wijst naar de bestemming, welke ook overal in het geheugen mag zitten.

De Z80 heeft de beschikking over vier blok verplaatsingen:

LDD, LDDR, LDI en LDIR.

Na iedere verplaatsing van een byte wordt door alle instructies BC met 1 verminderd. LDD en LDDR verlagen daarna DE en HL ieder met 1. LDI en LDIR verhogen DE en HL ieder met 1. Achter twee instructies staat een R van repeat (herhaal). Laten we de instructies eens nader bekijken.

LDI is de afkorting van "load and increment", wat "laad en tel 1 op" betekent. De instructie verplaatst een byte van het geheugen adres dat in HL staat naar het geheugen adres dat in DE staat. Daarna wordt BC met 1 verminderd. HL en DE worden automatisch al met 1 verhoogd, zodat ze al naar de volgende adressen wijzen.

LDIR betekent "load increment and repeat", d.w.z. "laad, tel 1 op, en herhaal dat". In feite wordt de instructie LDI uitgevoerd, totdat BC de waarde 0 heeft. M.b.v. deze instructie kan automatisch een heel blok data in het geheugen verplaatst worden.

LDD en LDDR werken op de zelfde manier, met dien verstande, dat DE en HL nu iedere keer met 1 worden verlaagd. De verplaatsing begint dus op het *hoogste* adres van het blok i.p.v. het laagste. Figuur 4.5 geeft een samenvatting van deze vier instructies.

Analoge geautomatiseerde instructies zijn beschikbaar voor CP (= compare = vergelijk). Zie figuur 4.6.

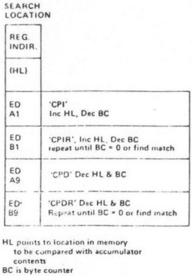


Fig. 4.6: Blok vergelijk groep-"CP"

Data verwerkende instructies

Rekenkundig

Er is voorzien in twee belangrijke rekenkundige bewerkingen: optellen en aftrekken. Ze zijn herhaaldelijk in het vorige hoofdstuk gebruikt. Er zijn twee soorten optellingen, met en zonder carry, ADD en ADC respectievelijk.
Voor de aftrekking geldt het zelfde: met en zonder carry, SUB en SBC.

Bovendien zijn de volgende instructies toegevoegd: DAA, CPL en NEG. DAA, "decimal adjust accumulator", wordt bij BCD bewerkingen gebruikt. Normaal komen ze voor in combinatie met iedere BCD optelling en aftrekking. Twee-complement instructies zijn ook beschikbaar. CPL berekent het een-complement van de accumulator, en NEG verandert het teken in het tegengestelde (in het twee-complement).

Alle voorgaande instructies werken met 8-bits data. 16-bits bewerkingen hebben meer beperkingen. ADD, ADC en SBC zijn alleen met

bepaalde registers te gebruiken. Zie figuur 4.8.

Tenslotte kunnen "increment" en "decrement" instructies gebruikt worden met alle registers, zowel in het 8-bits als in het 16-bits formaat. Zie de figuren 4.7 (8-bits) en 4.8 (16-bits).

BRON

	REGISTER ADDRESSING							REG. INDIR.	INDEXED		IMMED.
	A	B	C	D	E	H	L	(HL)	(IX+d)	(IY+d)	n
'ADD'	87	80	81	82	83	84	85	86	DD 86 d	FD 86 d	C6 n
ADD w CARRY 'ADC'	8F	88	89	8A	8B	8C	8D	8E	DD 8E d	FD 8E d	CE n
SUBTRACT 'SUB'	97	90	91	92	93	94	95	96	DD 96 d	FD 96 d	D6 n
SUB w CARRY 'SBC'	9F	98	99	9A	9B	9C	9D	9E	DD 9E d	FD 9E d	DE n
'AND'	A7	A0	A1	A2	A3	A4	A5	A6	DD A6 d	FD A6 d	E6 n
'XOR'	AF	A8	A9	AA	AB	AC	AD	AE	DD AE d	FD AE d	EE n
'OR'	B7	B0	B1	B2	B3	B4	B5	B6	DD B6 d	FD B6 d	F6 n
COMPARE 'CP'	BF	B8	B9	BA	BB	BC	BD	BE	DD BE d	FD BE d	FE n
INCREMENT 'INC'	3C	04	0C	14	1C	24	2C	34	DD 34 d	FD 34 d	
DECREMENT 'DEC'	3D	05	0D	15	1D	25	2D	35	DD 35 d	FD 35 d	

Fig. 4.7: 8-bits rekenkundige en logische bewerkingen

Alle rekenkundige instructies, behalve INC en DEC, veranderen een of meer status bits. In de bijlage wordt dit volledig beschreven. Dit detail moeten we heel goed onthouden. INC en DEC ... bij registerparen... zullen het Z-bit in het F register nooit 1 maken. Is dat in een programma nodig, dan moet dat expliciet gedaan worden.

Ook belangrijk is, dat ADC en SBC alle vlaggen beinvloeden. Dat betekent niet, dat na deze instructies alle vlaggen noodzakelijk een andere waarde zullen hebben. Het is echter wel mogelijk.

BESTEMMING		BRON: BC	DE	HL	SP	IX	IY
'ADD'	HL	09	19	29	39		
	IX	DD 09	DD 19		DD 39	DD 29	
	IY	FD 09	FD 19		FD 39		FD 29
ADD WITH CARRY AND SET FLAGS 'ADC'	HL	ED 4A	ED 5A	ED 6A	ED 7A		
SUB WITH CARRY AND SET FLAGS 'SBC'	HL	ED 42	ED 52	ED 62	ED 72		
INCREMENT 'INC'		03	13	23	33	DD 23	FD 23
DECREMENT 'DEC'		0B	1B	2B	3B	DD 2B	FD 2B

Fig. 4.8: 16-bits rekenkundige en logische bewerkingen

Logische bewerkingen

Er zijn drie logische instructies: AND, OR en XOR, plus een vergelijk instructie CP. Ze werken alle met 8 bit data. Een tabel met alle mogelijkheden en opcodes van deze instructies staat in figuur 4.7.

AND

Iedere logische bewerking wordt beschreven met een *waarheids tabel*. Daarin worden de logische uitkomsten van iedere mogelijke ingangs situatie gegeven. Hier volgt de waarheids tabel van de AND (EN):

0 AND 0 = 0
0 AND 1 = 0
1 AND 0 = 0
1 AND 1 = 1

of

en	0	1
0	0	0
1	0	1

De uitkomst van een AND functie is alleen 1 als beide ingangen 1 zijn. M.a.w. als een van de ingangen 0 is, dan is de uitkomst 0. De functie kan gebruikt worden om een bepaald bit in een woord 0 te maken. Dat is maskeren. Stel dat we de vier rechtse bits van een woord 0 willen maken. Dat kan gedaan worden met het volgende programma:

```
LD     A, WORD       WORD = "10101010"
AND    11110000B     "11110000" IS HET MASKER
```

De B geeft aan dat het een binair getal is.

Oefening 4.1: Schrijf een drie regels groot programma, dat bits 1 en 6 van WORD nul maakt.

Oefening 4.2: Wat gebeurt er als het masker "11111111" is?

OR

Allereerst de waarheids tabel:

0 OR 0 = 0
0 OR 1 = 1
1 OR 0 = 1
1 OR 1 = 1

of

of	0	1
0	0	1
1	1	1

Is een van de ingangen 1, dan is de uitkomst altijd 1. M.b.v. deze instructie kunnen we dus bepaalde bits in een woord 1 maken. Stel de vier rechtse bits van WORD moeten 1 worden:

```
LD   A,WORD
OR   00001111B
```

Als WORD de inhoud "10101010" heeft, dan wordt deze "10101111".

Oefening 4.3: Wat gebeurt er als we de instructie OR 10101111B zouden gebruiken in het vorige programma?

Oefening 4.4: Wat is het effect van een OR met "FF" hexadecimaal?

XOR

XOR betekent exclusieve OR. Het resultaat van de functie is 1, als een ingang en niet meer dan een ingang gelijk is aan 1. Zijn beide ingangen 1, dan is de uitkomst 0.

0 XOR 0 = 0
0 XOR 1 = 1
1 XOR 0 = 1
1 XOR 1 = 0

of

XOR	0	1
0	0	1
1	1	0

De XOR is uitstekend geschikt voor vergelijkingen. Zijn twee bits verschillend, dan is de uitkomst van een XOR van deze bits 1. Bovendien kan de instructie gebruikt worden om een woord te complementeren. Dat wordt door het volgende programma gedaan:

```
LD    A,WORD
XOR   11111111B
```

Stel WORD is "10101010", dan is de uitkomst "01010101". Dat is het complement van de oorspronkelijke waarde.

Oefening 4.5: Wat is het effect van een XOR met "00H"?

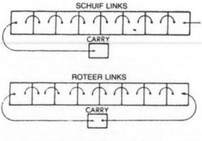


Fig. 4.9: Schuiven en roteren

Schuiven en roteren

Allereerst nog even het verschil tussen schuiven en roteren. Zie figuur 4.9. De schuif instructie verschuift de inhoud van een register een bit positie naar links of naar rechts. Het uitgeschoven bit komt in de carry terecht, en het binnenkomende bit is 0. Dit is allemaal al uitgelegd.

Er is een uitzondering: de rekenkundige verschuiving naar rechts. Bij getallen in het twee-complement, is het linkse bit het tekenbit. Dit bit is bij negatieve getallen 1. Delen we zo'n getal door 2 door het naar rechts te schuiven, dan moet het negatief blijven. D.w.z. het linkse bit moet 1 blijven. Dat doet de SRA instructie automatisch. Bij deze rekenkundige verschuiving naar rechts is het binnenkomende bit gelijk aan het tekenbit. Is het tekenbit 0, dan wordt een 0 naar binnen geschoven, is het een 1, dan een 1. In figuur 4.10 is dit geillustreerd.

Rotatie

Bij de rotatie is het binnenschuivende bit gelijk aan het bit dat uit de carry, of hetzelfde register valt. Dit bit gaat dus niet verloren zoals bij de schuif instructies. De Z80 heeft twee soorten rotaties: de 8-bits en de 9-bits rotatie.

Figuur 4.11 illustreert de 9-bits rotatie. In het geval van een rotatie naar rechts, worden de 8 bits van het register naar rechts geschoven.

Source and Destination

TYPE OF ROTATE OR SHIFT	A	B	C	D	E	H	L	(HL)	(IX + d)	(IY + d)
'RLC'	CB 07	CB 00	CB 01	CB 02	CB 03	CB 04	CB 05	CB 06	DD CB d 06	FD CB d 06
'RRC'	CB 0F	CB 08	CB 09	CB 0A	CB 0B	CB 0C	CB 0D	CB 0E	DD CB d 0E	FD CB d 0E
'RL'	CB 17	CB 10	CB 11	CB 12	CB 13	CB 14	CB 15	CB 16	DD CB d 16	FD CB d 16
'RR'	CB 1F	CB 18	CB 19	CB 1A	CB 1B	CB 1C	CB 1D	CB 1E	DD CB d 1E	FD CB d 1E
'SLA'	CB 27	CB 20	CB 21	CB 22	CB 23	CB 24	CB 25	CB 26	DD CB d 26	FD CB d 26
'SRA'	CB 2F	CB 28	CB 29	CB 2A	CB 2B	CB 2C	CB 2D	CB 2E	DD CB d 2E	FD CB d 2E
'SRL'	CB 3F	CB 38	CB 39	CB 3A	CB 3B	CB 3C	CB 3D	CB 3E	DD CB d 3E	FD CB d 3E
'RLD'								ED 6F		
'RRD'								ED 67		

	A
RLCA	07
RRCA	0F
RLA	17
RRA	1F

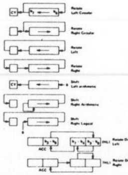


Fig. 4.10: Rotaties en verschuivingen

Het bit, dat rechts uit het register valt, komt in het carry bit. Het bit dat het register links binnenkomt, heeft de vorige waarde van het carry bit. Het is een 9-bits rotatie: het register plus het carry bit doen eraan mee. De rotatie naar rechts doet precies het zelfde, maar dan in tegengestelde richting.

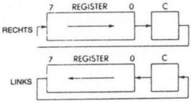
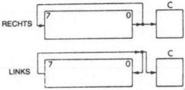


Fig. 4.11: Negen-bits rotatie

De 8-bits rotatie werkt op een ongeveer gelijke wijze. Bit 0 komt in bit 7 terecht, of omgekeerd, afhankelijk van de richting waarin geschoven wordt. Bovendien wordt het naar buiten geschoven bit gecopieerd in het carry bit. Zie figuur 4.12.



Figuur 4.12: Acht-bits rotatie

Speciale BCD schuif instructies

Er zijn twee roteer instructies speciaal om te gebruiken bij BCD getallen. Het zijn vier-bits rotaties tussen het lage orde deel van de accumulator en het geheugen adres waar HL naar wijst. Zie figuur 4.13.

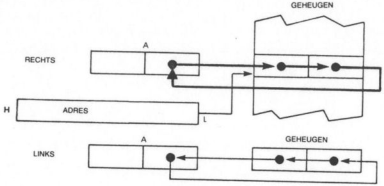


Fig. 4.13: BCD roteer instructies

Bit manipulatie

We heben gezien hoe m.b.v. logische functies bits in bepaalde registers 1 of 0 gemaakt kunnen worden. Dat zouden we echter graag in alle registers en in het geheugen willen doen. Daarvoor zijn veel opcodes nodig, en daarom is dit soort instructies meestal afwezig bij microprocessors. De Z80 heeft ze wel. Ze worden getoond in figuur 4.14. In deze tabel staan ook de test instructies, die besproken worden in de volgende paragraaf.

Twee instructies zijn beschikbaar, die een bewerking uitvoeren op het carry bit: CCF, welke de carry complementeert, en SCF, welke het carry bit 1 maakt. Zie figuur 4.15.

Test en spring

Aangezien tests erg afhankelijk zijn van het gebruik van het vlaggen of status register, zullen we de rol van iedere vlag hier gedetailleerd behandelen. Figuur 4.16 geeft de inhoud van het status register.

C is de carry, N is optellen of aftrekken, P/V is de pariteit of overflow, Z is nul, S is teken. De bits 3 en 5 worden niet gebruikt, en zijn altijd 0. H en N worden gebruikt bij BCD rekenen, en kunnen niet worden getest. De andere vier vlaggen (C, P/V, Z en S) kunnen getest worden in samenhang met call en sprong instructies.

Iedere vlag zal nu worden beschreven.

		REGISTER ADDRESSING							REG. INDIR.	INDEXED	
	BIT	A	B	C	D	E	H	L	(HL)	(IX+d)	(IY+d)
TEST 'BIT'	0	CB 47	CB 40	CB 41	CB 42	CB 43	CB 44	CB 45	CB 46	DD CB d 46	FD CB d 46
	1	CB 4F	CB 48	CB 49	CB 4A	CB 4B	CB 4C	CB 4D	CB 4E	DD CB d 4E	FD CB d 4E
	2	CB 57	CB 50	CB 51	CB 52	CB 53	CB 54	CB 55	CB 56	DD CB d 56	FD CB d 56
	3	CB 5F	CB 58	CB 59	CB 5A	CB 5B	CB 5C	CB 5D	CB 5E	DD CB d 5E	FD CB d 5E
	4	CB 67	CB 60	CB 61	CB 62	CB 63	CB 64	CB 65	CB 66	DD CB d 66	FD CB d 66
	5	CB 6F	CB 68	CB 69	CB 6A	CB 6B	CB 6C	CB 6D	CB 6E	DD CB d 6E	FD CB d 6E
	6	CB 77	CB 70	CB 71	CB 72	CB 73	CB 74	CB 75	CB 76	DD CB d 76	FD CB d 76
	7	CB 7F	CB 78	CB 79	CB 7A	CB 7B	CB 7C	CB 7D	CB 7E	DD CB d 7E	FD CB d 7E
RESET BIT 'RES'	0	CB 87	CB 80	CB 81	CB 82	CB 83	CB 84	CB 85	CB 86	DD CB d 86	FD CB d 86
	1	CB 8F	CB 88	CB 89	CB 8A	CB 8B	CB 8C	CB 8D	CB 8E	DD CB d 8E	FD CB d 8E
	2	CB 97	CB 90	CB 91	CB 92	CB 93	CB 94	CB 95	CB 96	DD CB d 96	FD CB d 96
	3	CB 9F	CB 98	CB 99	CB 9A	CB 9B	CB 9C	CB 9D	CB 9E	DD CB d 9E	FD CB d 9E
	4	CB A7	CB A0	CB A1	CB A2	CB A3	CB A4	CB A5	CB A6	DD CB d A6	FD CB d A6
	5	CB AF	CB A8	CB A9	CB AA	CB AB	CB AC	CB AD	CB AE	DD CB d AE	FD CB d AE
	6	CB B7	CB B0	CB B1	CB B2	CB B3	CB B4	CB B5	CB B6	DD CB d B6	FD CB d B6
	7	CB BF	CB B8	CB B9	CB BA	CB BB	CB BC	CB BD	CB BE	DD CB d BE	FD CB d BE
SET BIT 'SET'	0	CB C7	CB C0	CB C1	CB C2	CB C3	CB C4	CB C5	CB C6	DD CB d C6	FD CB d C6
	1	CB CF	CB C8	CB C9	CB CA	CB CB	CB CC	CB CD	CB CE	DD CB d CE	FD CB d CE
	2	CB D7	CB D0	CB D1	CB D2	CB D3	CB D4	CB D5	CB D6	DD CB d D6	FD CB d D6
	3	CB DF	CB D8	CB D9	CB DA	CB DB	CB DC	CB DD	CB DE	DD CB d DE	FD CB d DE
	4	CB E7	CB E0	CB E1	CB E2	CB E3	CB E4	CB E5	CB E6	DD CB d E6	FD CB d E6
	5	CB EF	CB E8	CB E9	CB EA	CB EB	CB EC	CB ED	CB EE	DD CB d EE	FD CB d EE
	6	CB F7	CB F0	CB F1	CB F2	CB F3	CB F4	CB F5	CB F6	DD CB d F6	FD CB d F6
	7	CB FF	CB F8	CB F9	CB FA	CB FB	CB FC	CB FD	CB FE	DD CB d FE	FD CB d FE

Fig. 4.14: Bit manipulatie groep

Decimal Adjust Acc, 'DAA'	27
Complement Acc, 'CPL'	2F
Negate Acc, 'NEG' (2's complement)	ED 44
Complement Carry Flag, 'CCF'	3F
Set Carry Flag, 'SCF'	37

Fig. 4.15: AF instructies

7	6	5	4	3	2	1	0
S	Z	-	H	-	P/V	N	C
(T)	(T)				(T)		(T)

Fig. 4.16: Het status register

Carry (C)

In bijna alle microprocessors, en zeker in de Z80, speelt de carry een dubbele rol. Allereerst wordt dit bit gebruikt om bij optellingen en aftrekkingen de carry of de borrow aan te geven. Ten tweede is het het negende bit bij schuif en roteer instructies. Door deze dubbele rol zijn enkele bewerkingen mogelijk, zoals de vermenigvuldiging. Dat moet duidelijk zijn door het voorbeeld in het vorige hoofdstuk.

Het is belangrijk te onthouden, dat alle rekenkundige instructies dit bit 0 of 1 maken, afhankelijk van de uitkomst van deze instructie. Schuif en roteer instructies hebben ook invloed op het carry bit.

Alle logische instructies (AND, OR en XOR) maken het carry bit 0. Deze instructies kunnen daar ook expliciet voor gebruikt worden.

De instructies, die het carry bit beinvloeden zijn: ADD A,s; ADC A,s; SUB s; SBC A,s; CP s; NEG; AND s; OR s; XOR s; ADD−DD,ss; ADC HL,ss; SBC HL,ss; RLA; RLCA; RRA; RRCA;

RL m; RLC m; RR m; RRC m; SLA m; SRA m; SRL m; DDA; SCF; CCF; NEG s.

Aftrekken (N)

Dit bit wordt normaal niet door de programmeur gebruikt, maar door de Z80 zelf bij BCD bewerkingen. In het vorige hoofdstuk is verteld, dat na iedere BCD optelling of aftrekking een DAA instructie wordt uitgevoerd om het resultaat te corrigeren. Deze correctie is bij een optelling anders als bij een aftrekking. Wat DAA doet is afhankelijk van de N vlag. N is 0 na een optelling, en 1 na een aftrekking. De afkorting is misschien misleidend voor degene die al gewerkt heeft met andere processors. Daar wordt N vaak gebruikt als teken bit.

N wordt 0 door: ADD A,s; ADC A,s; AND s; OR s; XOR s; INC s; ADD DD,ss; ADC HL,ss; RLA; RLCA; RRA; RRCA; RL m; RLC m; RR–m; RRC m; SLA m; SRA m; SRL m; RLD; RRD; SCF; CCF; IN r,(C); LDI; LDD; LDIR; LDDR; LD A,I; LD A,r; BIT B,s.

N wordt 1 door: SUB s; SBC A,s; CP s; NEG; DEC m; SBC HL,ss; CPL; INI; IND; OUTI; OUTD; INIR; INDR; OTIR; OTDR; CPI; CPIR; CPD en CPDR.

Pariteit/overflow

De pariteit/overflow vlag heeft twee functies. Bepaalde instructies setten (1 maken) of resetten (0 maken) deze vlag, afhankelijk van de pariteit van het resultaat. De pariteit wordt bepaald, door het aantal enen van het resultaat te tellen. Is dit aantal oneven, dan wordt de vlag 0 (oneven pariteit), is het aantal enen een even getal, dan wordt de pariteit vlag 1 gemaakt. Pariteit wordt het meest bij blokken karakters (meestal ASCII) gebruikt. Het pariteits bit wordt dan toegevoegd aan de zeven bits ASCII code ter controle of geen bit per ongeluk is veranderd. Stel er is een bit veranderd door een fout in het geheugen (b.v. RAM of disk), of tijdens het overseinen van het karakter, dan is het aantal enen in het karakter veranderd. Door het pariteits bit te controleren kan ontdekt worden, dat de code fout is. De vlag wordt gebruikt bij schuif en roteer instructies, en natuurlijk ook bij in- of uitvoer van of naar een randapparaat.

De 8080 gebruikt deze vlag uitsluitend om de pariteit aan te geven. De Z80 gebruikt de vlag echter ook voor andere dingen. Als je overstapt van de ene microprocessor naar de andere moet je dan ook altijd voorzichtig zijn met het gebruiken van deze vlag.

De Z80 gebruikt dit bit ook als overflow vlag.(de 8080 doet dat niet!). Bij de behandeling van het twee-complement in hoofdstuk 1 hebben we al kennis gemaakt met de overflow vlag. De vlag geeft aan, dat tijdens een optelling of een aftrekking het tekenbit "per ongeluk" is veranderd doordat het resultaat te groot is geworden. (M.b.v. acht bits is +127 het grootste en −128 het kleinste getal in het twee-complement).

Het bit wordt bovendien nog voor twee verschillende functies gebruikt.

Tijdens blok overdrachten (LDD, LDDR, LDI en LDIR), en tijdens zoek instructies (CPD, CPDR, CPI en CPIR) geeft het bit aan, of register B de waarde 0 heeft bereikt. Bij "decrement" instructies wordt de vlag 0, als het teller register paar 0 is. Bij "increment" instructies wordt de vlag 1, als aan het begin van de instructie BC − 1 = 0.

Tenslotte geeft de vlag bij de instructies LD A,I en LD A,R de waarde van de "interrupt enable flip-flop" (IFF2) weer. Op deze wijze kan de inhoud van de flip-flop getest worden.

De P vlag wordt beinvloed door: AND s; OR s; XOR s; RL m; RLC m; RR m; RRC m; SLA m; SRA m; SRL m; RLD; RRD; DAA; en IN r,(C).

De V vlag is afhankelijk van: ADD A,s; ADC A,s; SUB s; SBC−A,s; CP s; NEG; INC s; DEC m; ADC HL,ss; SBC HL,ss; NEG.

De vlag wordt ook gebruikt door: LDIR; LDDR (0 maken); LDI; LDD; CPI; CPIR; CPD; CPDR.

De half-carry vlag

Deze vlag geeft tijdens rekenkundige bewerkingen een mogelijke carry van bit 3 naar bit 4 aan. M.a.w. het geeft een carry aan van de minst significante nibble naar de meest significante. Duidelijk is, dat deze vlag primair bedoeld is voor BCD instructies. In het bijzonder wordt de vlag intern door de processor gebruikt bij de DAA instructie.

De vlag wordt 1, als er tijdens een optelling een carry is tussen bit 3 en 4. Is er geen carry, dan wordt de vlag 0. Tijdens een aftrekking wordt de vlag 1 als er een borrow is tussen bit 4 en 3. De vlag wordt 0 als er geen borrow is.

De vlag wordt beinvloed door een optelling, aftrekking, increment, decrement, vergelijking en logische bewerking. Dat is door de volgende instructies: ADD A,r; ADD A,s; SUB s; SBC A,s; CP s; NEG; AND s; OR s; XOR s; INC s; DEC s; RLA; RLCA; RRA; RRCA; RL

m; RLC m; RR m; RRC m; SLA m; SR m; SRL m; RLD; RRD; DAA; CPL; SCF; IN r,(C); LDI; LLD; LDIR; LDDR; LD A; LD A,r; BIT b,r; NEG s.

NB: de 16-bits optelling en aftrekking beinvloeden deze vlag niet.

Zero (Z)

Deze vlag geeft aan, of de waarde van een berekend of verplaatst byte 0 is. Zero wordt eveneens gebruikt bij vergelijkingen en nog enkele andere functies.

Is het resultaat van een bepaalde bewerking of data verplaatsing 0, dan wordt het zero bit 1 gemaakt. Zero wordt anders 0.

Bij vergelijkingen wordt zero 1 als de vergelijking opgaat. Z wordt gereset in het andere geval.

De Z80 gebruikt dit bit nog voor drie andere doeleinden: Z wordt geset tot 1, als tijdens een BIT instructie het gespecificeerde bit 0 is.

Bij de "blok input/output instructies" (INI, IND, OUTI, OUTD) wordt Z geset als $D - 1 = 0$. Bij INIR, INDR, OTIR en OTDR wordt de vlag geset als de byte teller 0 wordt.

Tenslotte wordt Z 1 als het input byte tijdens IN r,(C) de waarde 0 heeft.

Samenvattend wordt Z beinvloed door de volgende instructies: ADD A,s; ADC A,s; SUB s; SBC A,s; CP s; NEG; AND s; OR s; XOR s; INC s; DEC s; ADC HL,ss; SBC HL,ss; RL m; RLC m; RR m; RRC m; SLA m; SRA m; SRL m; RLD; RRD; DAA; IN r,(C); INI; IND; OUTI; OUTD; INIR; INDR; OTIR; OTDR; CPI; CPIR; CPD; CPDR; LD A,I; LD A,R; BIT b,s; NEG s.

Gebruikelijke instructies die geen invloed op Z hebben zijn: ADD DD,ss; RLA; RLCA; RRA; RRCA; CPL; SCF; CCF; LDI; LDD; LDIR; LDDR; INC DD; DEC DD.

Teken (S)

Deze vlag is een copie van het meest significante bit van een resultaat of van een verplaatst byte. In de twee-complement notatie is dat bit het teken bit. "0" geeft een positief teken weer, en "1" een negatief.

Bij de meeste microprocessors speelt dit bit een belangrijke rol, vooral bij in- en output instructies. Deze processors zijn in de meeste gevallen niet uitgerust met een BIT instructie, waarmee ieder bit in het geheugen getest kan worden. Het tekenbit is in die gevallen het gemakkelijkst te testen bit. Het lezen van de status van een input/output ap-

paraat beinvloedt het tekenbit van de processor. Dit bit wordt gelijk aan bit 7 van het status register. Het tekenbit kan gemakkelijk door het programma worden getest. Daarom bevindt de belangrijkste indicator van I/O chips (klaar/niet klaar) zich meestal op bit positie 7.

De Z80 heeft een speciale BIT instructie. Om een geheugen locatie (dat mag ook een I/O status register zijn) te testen, moet eerst het adres geladen worden in IX, IY, of HL. Een bepaald geheugen adres kan niet direct getest worden. Ook in het geval van de Z80 is het dus waardevol bit 7 te gebruiken als klaar/niet klaar vlag.

Instructies die het teken bit beinvloeden zijn: ADD A,s; SUB s; SBC A,s; CP s; NEG; AND s; OR s; XOR s; INC s; DEC m; ADC HL,ss; SBC HL,ss; RL m; RLC m; RR m; RRC m; SLA m; SRA m; SRL m; RLD; RRD; DAA; IN r,(C); CPR; CPIR; CPD; CPDR; LD A,I; LD A,r; NEG.

Samenvatting van de vlaggen

De vlaggen of status bits worden gebruikt om speciale condities te ontdekken binnen de ALU van de microprocessor. Ze kunnen m.b.v. speciale instructies getest worden, zodat de juiste actie ondernomen kan worden. Het is belangrijk de betekenis van de vlaggen te kennen, omdat de meeste beslissingen in een programma genomen worden op grond van deze vlaggen. Sprongen worden genomen naar adressen die afhangen van de waarden van de vlaggen. De enige uitzondering daarop is het interrupt mechanisme, dat in het hoofdstuk over I/O besproken zal worden.

Nu is het belangrijk de hoofdfuncties van alle vlaggen te kennen. Tijdens het maken van programma's kan altijd de bijlage van dit boek worden geraadpleegd, om het juiste effect van de instructies op de vlaggen te weten te komen. In de meeste gevallen kunnen de vlaggen genegeerd worden, en de lezer die de vlaggen nu nog complex vindt, moet zich daardoor niet bang laten maken. De vlaggen en hun gebruik zullen steeds duidelijker worden, naarmate we meer programma's maken.

Figuur 4.17 geeft een samenvatting van de condities waarop de vlaggen geset en gereset worden.

De sprong instructies

Een sprong instructie dwingt een sprong naar een gespecificeerd adres af. Het verandert de normale loop van het programma. I.p.v. het

programma sequentieel te doorlopen, worden dan verschillen delen van het programma uitgevoerd. Sprongen kunnen voorwaardelijk en onvoorwaardelijk zijn. Een onvoorwaardelijk sprong is een sprong die gemaakt wordt onafhankelijk van welke conditie dan ook.

Een voorwaardelijke sprong wordt slechts gemaakt, als voldaan is aan een of meerdere condities. Dit type sprong wordt gebruikt voor het maken van beslissingen, die afhangen van bepaalde data of een berekend resultaat.

Deze voorwaardelijke sprongen zijn afhankelijk van de vlaggen, reden waarom deze eerst zijn behandeld. We kunnen nu de verschillende spronginstructies van de Z80 bekijken.

Er zijn twee soorten sprongen: sprongen binnen het hoofdprogramma (deze worden "jumps" genoemd), en sprongen van en naar subroutines (de return en de call). Het resultaat van iedere sprong instructie is, dat PT met een nieuw adres wordt geladen, en het programma gaat verder op dat adres.

De volle betekenis van sprongen kan slechts worden begrepen als we de verschillende adresserings mogelijkheden kennen van de microprocessor. Dat deel stellen we echter nog even uit tot het volgende hoofdstuk. We zullen ons daarom hier beperken tot de andere aspecten van de sprong.

Sprongen kunnen voorwaardelijk en onvoorwaardelijk zijn. Bij een voorwaardelijke sprong kan een van de vier vlaggen getest worden: Z, C, P/V en S vlag. Elke vlag kan getest worden op de waarde 0 of de waarde 1.

De gebruikte afkortingen zijn:

Z = nul (Z = 0)
NZ = niet nul (Z = 1)
C = carry (C = 1)
NC = geen carry (C = 0)
PO = oneven pariteit
PE = even pariteit
P = positief (S = 0)
M = negatief (S = 1)

De Z80 heeft nog een gecombineerde instructie, die we al vaak gebruikt hebben. De instructie trekt 1 af van register B en springt, zolang B ongelijk aan 0 is. Het is de DJNZ instructie.

CALL en RETURN kunnen ook voorwaardelijk of onvoorwaardelijk zijn. Ze testen op de zelfde wijze de vlaggen als de jumps.

INSTRUCTION	C	Z	P/V	S	N	H	COMMENTS
ADD A, s; ADC A, s	:	:	V	:	0	:	8-bit add or add with carry
SUB s; SBC A, s, CP s, NEG	:	:	V	:	1	:	8-bit subtract, subtract with carry, compare and negate accumulator
AND s	0	:	P	:	0	1	Logical operations
OR s, XOR s	0	:	P	:	0	0	And sets different flags
INC s	•	:	V	:	0	:	8-bit increment
DEC m	•	:	V	:	1	:	8-bit decrement
ADD DD, ss	:	•	•	•	0	X	16-bit add
ADC HL, ss	:	:	V	:	0	X	16-bit add with carry
SBC HL, ss	:	:	V	:	1	X	16-bit subtract with carry
RLA; RLCA, RRA, RRCA	:	•	•	•	0	0	Rotate accumulator
RL m, RLC m, RR m; RRC m SLA m; SRA m, SRL m	:	:	P	:	0	0	Rotate and shift location m
RLD, RRD	•	:	P	:	0	0	Rotate digit left and right
DAA	:	:	P	:	•	:	Decimal adjust accumulator
CPL	•	•	•	•	1	1	Complement accumulator
SCF	1	•	•	•	0	0	Set carry
CCF	:	•	•	•	0	X	Complement carry
IN r, (C)	•	:	P	:	0	0	Input register indirect
INI; IND; OUTI; OUTD	•	:	X	X	1	X	Block input and output
INIR; INDR; OTIR; OTDR	•	1	X	X	1	X	Z = 0 if B ≠ 0 otherwise Z = 1
LDI, LDD	•	X	:	X	0	0	Block transfer instructions
LDIR, LDDR	•	X	0	X	0	0	P/V = 1 if BC ≠ 0, otherwise P/V = 0
CPI, CPIR, CPD, CPDR	•	:	:	:	1	X	Block search instructions Z = 1 if A = (HL), otherwise Z = 0 P/V = 1 if BC ≠ 0, otherwise P/V = 0
LD A, I; LD A, R	•	:	IFF	:	0	0	The content of the interrupt enable flip-flop (IFF) is copied into the P/V flag
BIT b, s	•	:	X	X	0	1	The complement of bit b of location is copied into the Z flag
NEG	:	:	V	:	1	:	Negate accumulator

The following notation is used in this table:

SYMBOL	OPERATION
C	Carry/link flag. C=1 if the operation produced a carry from the MSB of the operand or result.
Z	Zero flag. Z=1 if the result of the operation is zero.
S	Sign flag. S=1 if the MSB of the result is one.
P/V	Parity or overflow flag. Parity (P) and overflow (V) share the same flag. Logical operations affect this flag with the parity of the result while arithmetic operations affect this flag with the overflow of the result. If P/V holds parity, P/V=1 if the result of the operation is even, P/V=0 if result is odd. If P/V holds overflow, P/V=1 if the result of the operation produced an overflow.
H	Half-carry flag. H=1 if the add or subtract operation produced a carry into or borrow from bit 4 of the accumulator.
N	Add/Subtract flag. N=1 if the previous operation was a subtract.
	H and N flags are used in conjunction with the decimal adjust instruction (DAA) to properly correct the result into packed BCD format following addition or subtraction using operands with packed BCD format.
:	The flag is affected according to the result of the operation
•	The flag is unchanged by the operation
0	The flag is reset by the operation.
1	The flag is set by the operation.
X	The flag is a "don't care."
V	P/V flag affected according to the overflow result of the operation.
P	P/V flag affected according to the parity result of the operation.
r	Any one of the CPU registers A, B, C, D, E, H, L.
s	Any 8-bit location for all the addressing modes allowed for the particular instruction.
ss	Any 16-bit location for all the addressing modes allowed for that instruction.
ii	Any one of the two index registers IX or IY.
R	Refresh counter.
n	8-bit value in range <0, 255>.
nn	16-bit value in range <0, 65535>
m	Any 8-bit location for all the addressing modes allowed for the particular instruction.

Fig. 4.17: Samenvatting van de vlaggen

Voorwaardelijke sprongen naar subroutines zijn zeer krachtige instructies, maar komen bij de meeste 8-bits microprocessors bijna niet voor. Ze verbeteren de efficientie van een programma, omdat nu in een instructie gedaan kan worden, waarvoor anders twee instructies nodig waren.

Tenslotte zijn er nog de instructies RETI en RETN, welke voorkomen in interrupt routines. Ze worden in hoofdstuk 6 besproken.

De adresserings mogelijkheden en de opcodes van de sprong instructies staan in figuur 4.18.

			CONDITION									
			UN-COND	CARRY	NON CARRY	ZERO	NON ZERO	PARITY EVEN	PARITY ODD	SIGN NEG	SIGN POS	REG B≠0
JUMP 'JP'	IMMED. EXT.	nn	C3 n n	DA n n	D2 n n	CA n n	C2 n n	EA n n	E2 n n	FA n n	F2 n n	
JUMP 'JR'	RELATIVE	PC+e	18 e-2	38 e-2	30 e-2	28 e-2	20 e-2					
JUMP 'JP'	REG. INDIR.	(HL)	E9									
JUMP 'JP'		(IX)	DD E9									
JUMP 'JP'		(IY)	FD E9									
'CALL'	IMMED. EXT.	nn	CD n n	DC n n	D4 n n	CC n n	C4 n n	EC n n	E4 n n	FC n n	F4 n n	
DECREMENT B, JUMP IF NON ZERO 'DJNZ'	RELATIVE	PC+e										10 e-2
RETURN 'RET'	REGISTER INDIR.	(SP) (SP+1)	C9	D8	D0	C8	C0	E8	E0	F8	F0	
RETURN FROM INT 'RETI'	REG. INDIR.	(SP) (SP+1)	ED 4D									
RETURN FROM NON MASKABLE INT 'RETN'	REG. INDIR.	(SP) (SP+1)	ED 45									

Fig. 4.18: Sprong instructies

De verschillende adresserings mogelijkheden worden in hoofdstuk 5 behandeld.

Uit figuur 4.18 blijkt, dat verschillende adresserings mogelijkheden een aantal beperkingen hebben. Bijvoorbeeld: de absolute sprong JP nn kan vier vlaggen testen, maar JR kan er slechts twee testen.

NB: JR wordt wanneer mogelijk verkozen boven JP, want de instructie is korter, en het programma kan eenvoudig in het geheugen

verplaatst worden. De instructies zijn echter niet altijd equivalent: JR kan de pariteit en teken vlaggen niet testen.

Er is nog een soort speciale sprong beschikbaar: De *restart* of RST instructie. Het is een een-bytes instructie, waarmee gesprongen kan worden naar een van de 8 start adressen aan het begin van het geheugen. Deze adressen zijn (decimaal): 0, 8, 16, 24, 32, 40, 48, en 56. Een krachtige instructie, aangezien hij een byte lang is. Het is de snelste sprong instructie die beschikbaar is, en daarom wordt hij het meest gebruikt als reactie op een interrupt. Voor andere doeleinden is hij echter ook te gebruiken. Zie figuur 4.19.

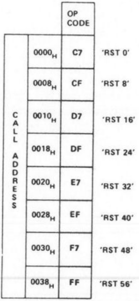


Fig. 4.19: RESTART groep

Input/output instructies

Input/output technieken worden behandeld in hoofdstuk 6. Eenvoudig kan gesteld worden, dat I/O apparatuur geadresseerd kan worden op twee manieren: als geheugen, m.b.v. de eerder besproken instruc-

ties, of m.b.v. speciale I/O instructies. Gewone geheugen adressering heeft drie bytes nodig: een byte voor de opcode en twee bytes voor het adres. Daardoor zijn ze langzaam. De speciale I/O instructies zijn korter en dus sneller. Ze hebben echter twee nadelen.

Ten eerste verspillen ze enkele van de waardevolle opcodes (voor de opcode worden 8 bits gebruikt en er is dus slechts een beperkt aantal combinaties mogelijk). Ten tweede genereren ze enkele speciale I/O signalen, waarvoor een paar (ook schaarse) IC pennen nodig zijn. Het aantal pennen is meestal beperkt tot 40. De meeste microprocessors hebben vanwege deze nadelen geen I/O instructies. De 8080 en dus de Z80 hebben ze wel.

Het voordeel van I/O instructies is dat ze sneller zijn, aangezien ze maar twee bytes lang zijn. Een zelfde resultaat kan behaald worden door een speciale adresseer techniek toe te passen: de "pagina 0" adressering. Het adres in de instructie is 8 bits groot. Het is een techniek, die vaak in andere processors wordt toegepast.

De twee basis I/O instructies zijn IN en OUT. Ze verplaatsen de inhoud van een werkregister naar een I/O apparaat, en omgekeerd. De instructies zijn twee bytes lang. Het eerste byte is de opcode. Het tweede byte is het minst significante deel van het adres. De accumulator bevat het meest significante deel van het adres. Daardoor kunnen 64K I/O apparaten geadresseerd worden. De accumulator moet echter eerst geladen worden met de juiste waarde. Dat kan het geheel langzaam maken.

De vier blok instructies voor input zijn: INI, INIR, IND en INDR. En voor output: OUTI, OTIR, OUTD en OTDR.
HL wordt gebruikt als pointer naar de bestemming*. In het geval van de output instructies verwijst de pointer in HL naar de oorsprong van de data. C selecteert het I/O apparaat (256 mogelijkheden). Register B wordt als teller gebruikt. Bij HL wordt 1 opgeteld als de instructie INI is, en er wordt 1 van afgetrokken als de instructie IND is.

INI verplaatst 1 byte, waarna bij HL 1 wordt opgeteld. B wordt met 1 verminderd.

INIR voert INI net zolang uit, totdat B 0 is. Zo kunnen tot 256 bytes worden verplaatst. B moet dan wel voor de instructie de waarde 0 gekregen hebben.

De figuren 4.20 en 4.21 geven een samenvatting van de opcodes.

Controle instructies

Controle instructies veranderen de werk mode van de CPU, of ver-

anderen zijn interne status. De Z80 heeft zeven van dergelijke instructies.

De NOP instructie doet totaal niets gedurende 1 cyclus. Hij wordt gebruikt voor het inbouwen van vertragingen (4 states = 2 microseconden met een 2 MHz klok). De gaten in een programma, ontstaan tijdens het foutzoeken kunnen ermee worden opgevuld. De opcode van NOP bestaat geheel uit nullen. Een lege geheugen plaats in het programma, ontstaan om welke reden dan ook, kan geen schade veroorzaken, omdat het dan in feite een NOP is. Ook laten ze het programma niet stoppen.

De HALT instructie wordt samen met interrupts of een reset gebruikt. HALT stopt de processor. Deze zal weer gaan werken na een interrupt of een reset. Tijdens de HALT mode blijft de processor NOP's uitvoeren. Tijdens het foutzoeken wordt meestal een halt achteraan het programma geplaatst, omdat het hoofdprogramma daarna toch niets meer heeft te doen. Het programma moet daarna wel weer expliciet worden opgestart.

Twee instructies setten of resetten de interne interrupt vlag: EI en DI. Interrupts worden in hoofdstuk 6 besproken. De interrupt vlag geeft aan of een interrupt wel of niet is toegestaan. Om interrupts te voorkomen tijdens gedeelten van het programma moet de interrupt vlag 0 gemaakt worden. In hoofdstuk 6 zal dat worden getoond. Figuur 4.22 laat deze instructies zien.

			SOURCE							
			REGISTER							REG. IND.
			A	B	C	D	E	H	L	(HL)
'OUT'	IMMED.	(n)	D3 n							
	REG. IND.	(C)	ED 79	ED 41	ED 49	ED 51	ED 59	ED 61	ED 69	
'OUTI' – OUTPUT Inc HL, Dec b	REG. IND.	(C)								ED A3
'OTIR' – OUTPUT, Inc HL, Dec B, REPEAT IF B≠0	REG. IND.	(C)								ED B3
'OUTD' – OUTPUT Dec HL & B	REG. IND.	(C)								ED AB
'OTDR' – OUTPUT, Dec HL & B, REPEAT IF B≠0	REG. IND.	(C)								ED BB

PORT DESTINATION ADDRESS (REG. IND. / (C) columns)

BLOCK OUTPUT COMMANDS ('OUTI', 'OTIR', 'OUTD', 'OTDR' rows)

Fig. 4.20: OUTPUT groep

SOURCE
PORT ADDRESS

INPUT DESTINATION

			IMMED. (n)	REG. INDIR. (C)
INPUT 'IN'	REG ADDRESSING	A	DB n	ED 78
		B		ED 40
		C		ED 48
		D		ED 50
		E		ED 58
		H		ED 60
		L		ED 68
'INI' – INPUT & Inc HL, Dec B	REG. INDIR	(HL)		ED A2
'INIR' – INP, Inc HL, Dec B, REPEAT IF B≠0				ED B2
'IND' – INPUT & Dec HL, Dec B				ED AA
'INDR' – INPUT Dec HL Dec B, REPEAT IF B≠0				ED BA

BLOCK INPUT COMMANDS ('INI', 'INIR', 'IND', 'INDR')

Fig. 4.21: INPUT groep

'NOP'	00	
'HALT'	76	
DISABLE INT '(DI)'	F3	
ENABLE INT '(EI)'	FB	
SET INT MODE 0 'IM0'	ED 46	8080A MODE
SET INT MODE 1 'IM1'	ED 56	CALL TO LOCATION 0038_H
SET INT MODE 2 'IM2'	ED 5E	INDIRECT CALL USING REGISTER I AND 8 BITS FROM INTERRUPTING DEVICE AS A POINTER.

Fig. 4.22: Diverse CPU controle instructies

De Z80 heeft drie verschillende soorten interrupts modes. De 8080 heeft er maar een. Soort 0 is de 8080 interrupt. Soort 1 is een call naar adres 038H, en soort 2 is een indirecte call, die gebruik maakt van de inhoud van een speciaal register I, plus 8 bits afkomstig van het apparaat dat de interrupt veroorzaakt, welke tesamen het adres van de interrupt routine vormen. In hoofdstuk 6 komt dit allemaal aan de orde.

Tenslotte heeft de Z80 een aantal pennen, waarmee ook een interrupt gegeven kan worden. Het zijn pennen BVSRQ en NMI.

SAMENVATTING

De vijf klassen beschikbare instructies van de Z80 zijn in dit hoofdstuk besproken. De volgende paragraaf geeft uitgebreide informatie over de afzonderlijke instructies. Je hoeft niet alle instructies te kennen, om te beginnen met programmeren. In het begin is kennis van de belangrijkste instructies genoeg. Om later echter efficiente programma's te kunnen maken is het wel belangrijk ze allemaal te kennen. In het begin is dat allemaal niet zo belangrijk, en kun je de meeste instructies gewoon negeren.

Een belangrijk aspect is tot nu toe niet behandeld: de verschillende adresseer mogelijkheden van de Z80. De adresserings technieken zullen we in het volgende hoofdstuk bestuderen.

BESCHRIJVING VAN DE Z80 INSTRUCTIES

AFKORTINGEN

VLAG	AAN	UIT
CARRY	C (CARRY)	NC(GEENCARRY)
TEKEN	M (MINUS)	P (PLUS)
NUL	Z (NUL)	NZ (NIET NUL)
PARITEIT	PE (EVEN)	PO (ONEVEN)

● vlag verandert overeenkomstig de bewerking
O vlag wordt nul
1 vlag wordt een
? vlag krijgt een willekeurige waarde
X speciaal geval, let op de voetnoten op de pagina

de bits 3 en 5 hebben altijd een willekeurige waarde

ADC A,s Tel accumulator en gespecificeerde operand op met carry

Funktie: A ← A + s + C

Formaat: s: Kan zijn r, n, (HL), (IX + d), of (IY + d)

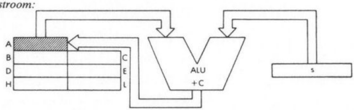
r	1 0 0 0 1 ← r →	
n	1 1 0 0 1 1 1 0	byte 1: CE
	← n →	byte 2: onmiddellijk data
(HL)	1 0 0 0 1 1 1 0	8E
(IX + d)	1 1 0 1 1 1 0 1	byte 1: DD
	1 0 0 0 1 1 1 0	byte 2: 8E
	← d →	byte 3: verplaatsing
(IY + d)	1 1 1 1 1 1 0 1	byte 1: FD
	1 0 0 0 1 1 1 0	byte 2: 8E
	← d →	byte 3: verplaatsing

R kan zijn:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Beschrijving: De operand s en de carry vlag uit het status register worden bij de accumulator opgeteld, en het resultaat komt in de accumulator. S wordt gedefinieerd in de gelijksoortige ADD instructie.

Datastroom:



Timing:

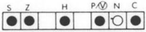
s:	*M cycli:*	*T states:*	*usec @ 2 MHz:*
r	1	4	2
n	2	7	3.5
(HL)	2	7	3.5
(IX + d)	5	19	9.5
(IY + d)	5	19	9.5

Adresseringsmode: r: implicit; n: onmiddellijk; (HL): indirect; (IX + d), (IY + d): Geïndexeerd.

Byte codes: ADC A,r

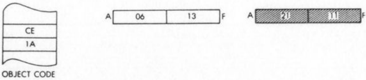
r:	A	B	C	D	E	H	L
	8F	88	89	8A	8B	8C	8D

Flags:



Voorbeeld: ADC A, 1A

Voor: Na:



ADC HL,ss

Tel HL en register paar ss met carry op

Funktie: HL ← HL + ss + C

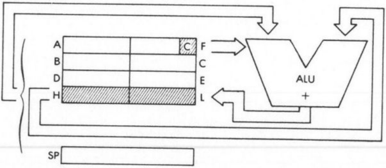
Formaat:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	s	s	1	0	1	0	byte 2

Beschrijving: De inhoud van HL wordt bij de inhoud van het gespecificeerde register paar opgeteld, daarna wordt de carry erbij opgeteld. Het resultaat wordt opgeslagen in HL. SS kan zijn:

BC - 00 HL - 10
DE - 01 SP - 11

Datastroom:



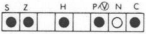
Timing: 4 M cycli; 15 T states: 75 usec @ 2 MHz

Adresseringsmode: Impliciet.

Byte codes: ss:

	BC	DE	HL	SP
ED-	4A	5A	6A	7A

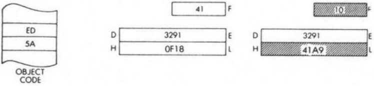
Flags:



H wordt geset door een carry van bit 11

Voorbeeld: ADC HL, DE

Voor: Na:



ADD A,(HL) Tel de accumulator op bij de indirect geadresseerde geheugen locatie (HL)

Funktie: A ← A + (HL)

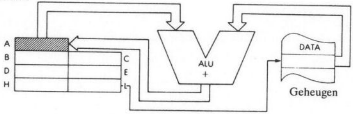
Formaat:

1	0	0	0	0	1	1	0

86

Beschrijving: De inhoud van de accumulator wordt opgeteld bij de inhoud van het adres, dat in HL staat. Het resultaat komt in de accumulator

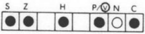
Datastroom:



Timing: 2 M cycli; 7 T states: 3.5 usec @ 2 MHz

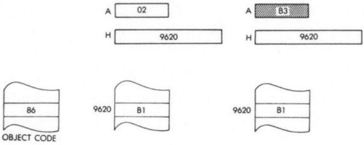
Adresseringsmode: Indirect.

Flags:



Voorbeeld: ADD A, (HL)

Voor: Na:



ADD A,(IX + d) Tel de accumulator op bij de geïndexeerd geadresseerde geheugen locatie (IX + d)

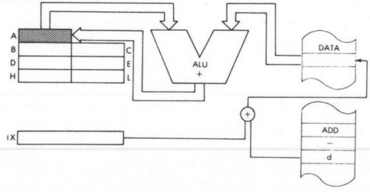
Funktie: A ← A + (IX + d)

Formaat:

1	1	0	1	1	1	0	1	byte 1: DD
1	0	0	0	0	1	1	0	byte 2: 86
← d →								byte 3: Verplaatsing

Beschrijving: De inhoud van de accumulator wordt opgeteld bij de inhoud van het geheugen adres, geadresseerd door de inhoud van het index register IX plus de verplaatsing

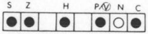
Datastroom:



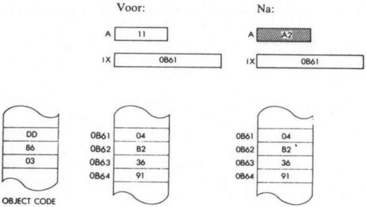
Timing: 5 M cycli; 19 T states: 9.5 usec @ 2 MHz

Adresseringsmode: Geïndexeerd.

Flags:



Voorbeeld: ADD A, (IX + 3)



ADD A,(IY + d) Tel de accumulator op bij de geindexeerd geadresseerde geheugen locatie (IY + d)

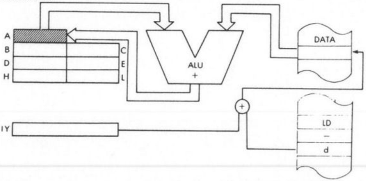
Funktie: A ← A + (IY + d)

Formaat:

1	1	1	1	1	1	0	1	byte 1: FD
1	0	0	0	0	1	1	0	byte 2: 86
←			d				→	byte 3: Verplaatsing

Beschrijving: De inhoud van de accumulator wordt opgeteld bij de inhoud van het geheugen adres, geadresseerd door de inhoud van het index register IY plus de verplaatsing.

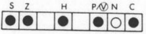
Datastroom:



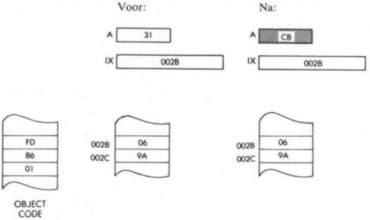
Timing: 5 M cycli; 19 T states: 9.5 usec @ 2 MHz

Adresseringsmode: Geïndexeerd.

Flags:



Voorbeeld: ADD A, (IY + 1)



ADD A,n

Tel de accumulator op bij immediate data n

Funktie: A ← A + n

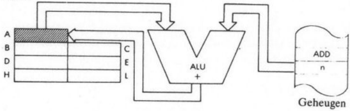
Formaat:

1	1	0	0	0	1	1	0	byte 1: C6

← n →	byte 2: onmiddellijk data

Beschrijving: De inhoud van de accumulator wordt opgeteld bij de inhoud van de geheugen locatie direct volgend op de code. Het resultaat komt in de accumulator

Datastroom:



Timing: 2 M cycli; 7 T states: 3.5 usec @ 2 MHZ

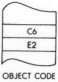
Adresseringsmode. Onmiddellijk.

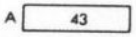
Flags:

S	Z		H		P/V	N	C
●	●		●		●	○	●

Voorbeeld: ADD A, E2

Voor: Na:



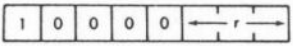




ADD A,r — Tel de accumulator op bij register r

Funktie: A ← A + r

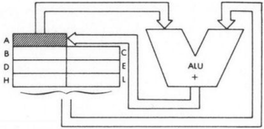
Formaat:



Beschrijving: De inhoud van de accumulator wordt opgeteld bij de inhoud van het gespecificeerde register. Het resultaat wordt in de accumulator opgeslagen. R mag zijn:

A - 111
B - 000
C - 001
D - 010
E - 011
H - 100
L - 101

Datastroom:



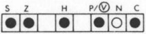
Timing: 1 M cyclus; 4 T states: 2 usec @ 2 MHz.

Adresseringsmode: Impliciet.

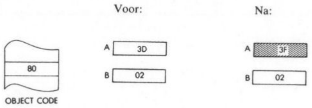
Byte codes:

A	B	C	D	E	H	L
87	80	81	82	83	84	85

Flags:



Voorbeeld: ADD A, B



ADD HL,ss

tel HL op bij register paar ss

Funktie: HL ← HL + ss

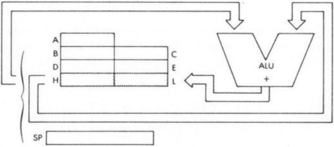
Formaat:

0	0	s	s	1	0	0	1

Beschrijving: De inhoud van HL wordt bij de inhoud van het register paar opgeteld, en het resultaat wordt in HL opgeslagen. SS kan zijn:

BC - 00	HL - 10
DE - 01	SP - 11

Datastroom:



Timing: 3 M cycli; 11 T states: 5.5 usec @ 2 MHz

Adresseringsmode: Impliciet.

Byte codes: ss:

BC	DE	HL	SP
09	19	29	39

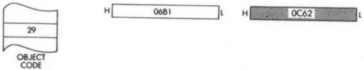
Flags:

S	Z		H		P/V	N	C
			●			○	●

C wordt geset door een carry van bit 15, en wordt anders gereset
H wordt geset door een carry van bit 11

Voorbeeld: ADD HL, HL

Voor: Na:



ADD IX,rr

Tel IX op bij register paar rr

Funktie: IX ← IX + rr

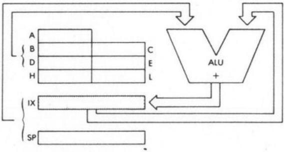
Formaat:

1	1	0	1	1	1	0	1	byte 1: DD
0	0	r	r	1	0	0	1	byte 2

Beschrijving: De inhoud van het IX register wordt opgeteld bij de inhoud van het gespecificeerde register paar. Het resultaat komt in IX. RR kan zijn:

BC - 00	IX - 10
DE - 01	SP - 11

Datastroom:



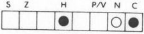
Timing: 4 M cycli; 15 T states: 7.5 usec @ 2 MHz

Adresseringsmode: Impliciet:

Byte codes:

rr:	BC	DE	IX	SP
DD-	09	19	29	39

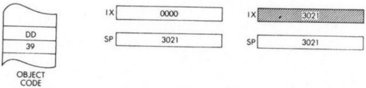
Flags:



H wordt geset door een carry van bit 11
C wordt geset door een carry van bit 15

Voorbeeld: ADD IX, SP

Voor: Na:



ADD IY,rr

Tel IY op bij register paar rr

Funktie: IY ← IY + rr

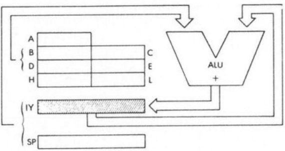
Formaat:

1	1	1	1	1	1	0	1	byte 1: FD
0	0	r	r	1	0	0	1	byte 2

Beschrijving: De inhoud van IY wordt opgeteld bij de inhoud van het gespecificeerde register paar, en het resultaat wordt in IY opgeslagen. RR mag zijn:

BC - 00	IY - 10
DE - 01	SP - 11

Datastroom:



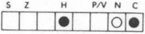
Timing: 4 M cycli; 15 T states: 7.5 usec @ MHz

Adresseringsmode: Impliciet:

Byte codes:

rr:	BC	DE	IY	SP
FD–	09	19	29	39

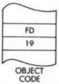
Flags:

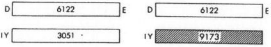


H wordt geset door een carry van bit 11
C wordt geset door een carry van bit 15

Voorbeeld: ADD IY, DE

Voor: Na:





AND s

Logische EN van accumulator en operand s

Funktie: A ← A ∧ s

Formaat: *s:* kan zijn r, n, (HL), (IX + d), of IY + d)

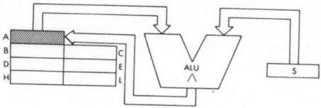
r	1 0 1 0 0 ←r→	
n	1 1 1 0 0 1 1 0	byte 1: E6
	←n→	byte 2: onmiddellijk data
(HL)	1 0 1 0 0 1 1 0	96
(IX + d)	1 1 0 1 1 1 0 1	byte 1: DD
	1 0 1 0 0 1 1 0	byte 2: 96
(IY + d)	←d→	byte 3: verplaatsing
	1 1 1 1 1 1 0 1	byte 1: FD
	1 0 1 0 0 1 1 0	byte 2: 96
	←d→	byte 3: verplaatsing

r kan een van de volgende zijn:

A – 111	E – 011
B – 000	H – 100
C – 011	L – 101
D – 010	

Beschrijving: De accumulator en de gespecificeerde operand worden logisch ge"en"d, en het resultaat komt in de accumulator. S wordt gedefinieerd in de ADD instructies

Datastroom:



Timing:

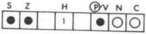
s:	M cycli:	T states:	usec @ 2 MHz:
r	1	4	2
n	2	7	3.5
(HL)	2	7	3.5
(IX + d)	5	19	9.5
(IY + d)	5	19	9.5

Adresseringsmode: r: impliciet; n: onmiddellijk; (HL): indirect; (IX + d), (IY + d): geïndexeerd.

Byte codes: AND r

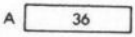
r:	A	B	C	D	E	H	L
	A7	A0	A1	A2	A3	A4	A5

Flags:



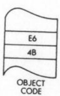
Voorbeeld: AND 4B

Voor:



Na:





BIT b,(HL)

Test bit b van de indirect geadresseerde geheugen locatie (HL)

Funktie: $Z \leftarrow \overline{(HL)_b}$

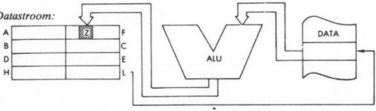
Formaat:

1	1	0	0	1	0	1	1	byte 1: CB
0	1	←	b	→	1	1	0	byte 2

Beschrijving: Bit b van de geheugen locatie geadresseerd door de inhoud van het HL register wordt getest, en de Z vlag wordt geset overeenkomstig het resultaat. B kan zijn:

0 - 000	4 - 100
1 - 001	5 - 101
2 - 010	6 - 110
3 - 011	7 - 111

Datastroom:



Timing: 3 M cycli; 12 T states: 6 usec @ 2 MHz

Adresseringsmode: Indirect.

Flags:

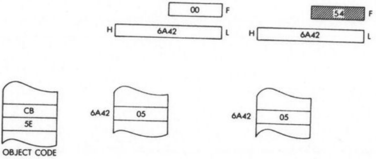
S	Z		H		P/V	N	C
?	●		1		?	0	

Byte codes:

b:	0	1	2	3	4	5	6	7
CB-	46	4E	56	5E	66	6E	76	7E

Voorbeeld: BIT 3, (HL)

Voor: Na:



BIT b,(IX + d) Test bit b van de geïndexeerd geadresseerde geheugen locatie (IX+d)

Funktie: $Z \leftarrow \overline{(IX + d)_b}$

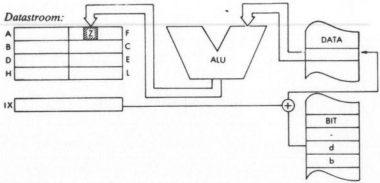
Formaat:

1	1	0	1	1	1	0	1	byte 1: DD
1	1	0	0	1	0	1	1	byte 2: CB
←			d				→	byte 3: verplaatsing
0	1	←	b	→	1	1	0	byte 4

Beschrijving: Het gespecificeerde bit van de geheugen locatie, geadresseerd door de inhoud van het IX register plus de gegeven verplaatsing, wordt getest, en de Z vlag wordt geset overeenkomstig het resultaat. B kan zijn:

0 - 000	4 - 100
1 - 001	5 - 101
2 - 010	6 - 110
3 - 011	7 - 111

Datastroom:



Timing: 5 M cycli; 20 T status: 10 usec @ 2 MHz

Adresseringsmode: Geïndexeerd:

Byte codes:

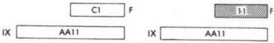
b:	0	1	2	3	4	5	6	7
DD-CB-d-	46	4E	56	5E	66	6E	76	7E

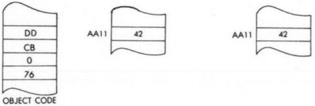
Flags:

S	Z		H		P/V	N	C
?	●		1		?	0	

Voorbeeld: BIT 6, (IX + 0)

Voor: Na:

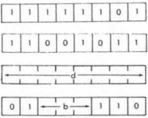




BIT b,(IY + d) Test bit b van de geindexeerd geadresseerde geheugen locatie (IY + d)

Funktie: $Z \leftarrow \overline{(IY + d)_b}$

Formaat:



byte 1: FD

byte 2: CB

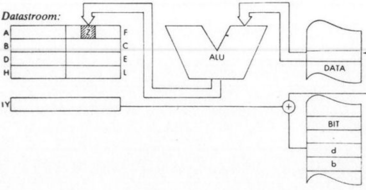
byte 3: verplaatsing

byte 4

Beschrijving: Het gespecificeerde bit van de geheugen locatie, geadresseerd door de inhoud van IY plus de verplaatsing, wordt getest, en de Z vlag wordt geset overeenkomstig het resultaat. B kan zijn:

0 - 000	4 - 100
1 - 001	5 - 101
2 - 010	6 - 110
3 - 011	7 - 111

Datastroom:



Timing: 5 M cycli; 20 T states: 10 usec @ 2 MHz

Adresseringsmode: Geïndexeerd.

Byte codes: b:

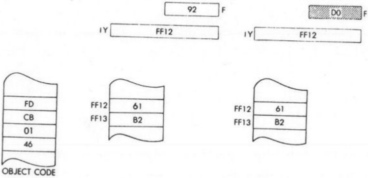
0	1	2	3	4	5	6	7
46	4E	56	5E	66	6E	76	7E

Flags:

S	Z		H		P/V	N	C
?	●		1		?	0	

Voorbeeld: BIT 0, (IY + 1)

Voor: Na:



BIT b,r Test bit b van register r

Funktie: $Z \leftarrow \overline{r_b}$

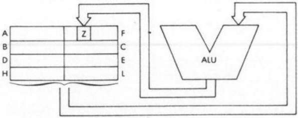
Formaat:

1	1	0	0	1	0	1	1	byte 1: CB
0	1	← b →			← r →			byte 2

Beschrijving: Het gespecificeerde bit van het gegeven register wordt getest, en de Z vlag wordt overeenkomstig het resultaat geset. B en r kunnen zijn:

b:	0 - 000	4 - 100
	1 - 001	5 - 101
	2 - 010	6 - 110
	3 - 011	7 - 111
r:	A - 111	E - 011
	B - 000	H - 100
	C - 001	L - 101
	D - 010	

Datastroom:



Timing: 2 M cycli; 8 T states: 4 usec @ 2 MHz

Adresseringsmode: Impliciet.

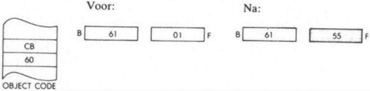
Byte codes:

b: \ r:	A	B	C	D	E	H	L
0	47	40	41	42	43	44	45
1	4F	48	49	4A	4B	4C	4D
2	57	50	51	52	53	54	55
3	5F	58	59	5A	5B	5C	5D
4	67	60	61	62	63	64	65
5	6F	68	69	6A	6B	6C	6D
6	77	70	71	72	73	74	75
7	7F	78	79	7A	7B	7C	7D

Flags:

S	Z		H		P/V	N	C
?	●		1		?	0	

Voorbeeld: BIT 4, B



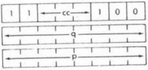
CALL cc,pq

Call subroutine op voorwaarde

Funktie:

indien cc waar: $(SP - 1) \leftarrow PC_{hoog}$ $(SP - 2) \leftarrow PC_{laag}$; $SP \leftarrow SP - 2$; $PC \leftarrow pq$
indien rr niet waar: $PC \leftarrow PC + 3$

Formaat:



byte 1
byte 2: adres, laag
byte 3: adres, hoog

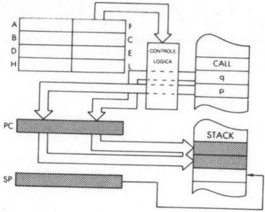
Beschrijving:

Als de voorwaarde waar is, dan wordt de inhoud van de programma teller op de stapel geplaatst, zoals beschreven is bij de PUSH instructie. Daarna wordt de inhoud van het adres volgend op de opcode in het lage orde deel van PC geladen, en de inhoud van het tweede byte volgend op de opcode wordt in het hoge orde deel van PC geladen. De volgende instructie wordt van het nieuwe adres gehaald. Wordt niet voldaan aan de voorwaarde, dan wordt adres pq genegeerd en de volgende instructie wordt uitgevoerd. CC kan zijn:

NZ - 000	PO - 100
Z - 001	PE - 101
NC - 010	P - 100
C - 011	M - 111

Een RET instructie kan gebruikt worden aan het eind van de subroutine, om de oude waarde van PC te hersyellen.

Datastroom:



Timing:

	M cycli:	*T states:*	*usec @ 2 MHz*
conditie waar:	5	17	8.5
conditie vals:	3	10	5

Adresseringsmode: Onmiddellijk

Byte codes:

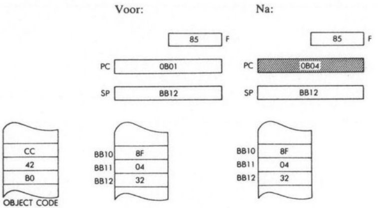
CC:	NZ	Z	NC	C	PO	PE	P	M	
	C4	CC	D4	DC	E4	EC	F4	FC	-q-p

Flags:

S	Z		H		P/V	N	C

(geen invloed).

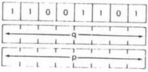
Voorbeeld: CALL Z, BO42



CALL pq Call subroutine op adres pq

Funktie: (SP − 1) ← PC_{hoog}; (SP − 2) ← PC_{laag}; SP ← − 2; PC ← pq

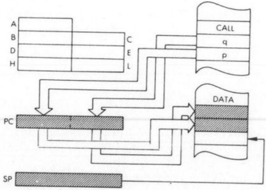
Formaat:



byte 1: CD
byte 2: adres, laag
byte 3: adres, hoog

Beschrijving: De inhoud van de programma teller wordt op de stapel geplaatst. Zie PUSH instructie. De inhoud van het adres volgend op de opcode komt in het lage orde deel van PC. De inhoud van het tweede byte volgend op de opcode komt in het hoge orde deel van PC. De volgende instructie komt van het nieuwe adres

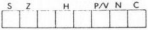
Datastroom:



Timing: 5 M cycli; 17 T states: 8.5 usec @ 2 MHz

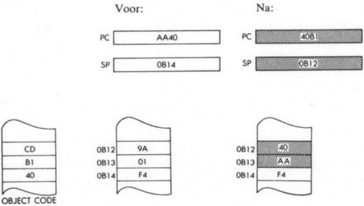
Adresseringsmode: Onmiddellijk

Flags:



(geen invloed).

Voorbeeld: CALL 40B1



CCF Complementeer de carry vlag

Funktie: $C \leftarrow \overline{C}$

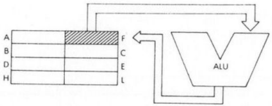
Formaat:

0	0	1	1	1	1	1	1

3F

Beschrijving: De carry vlag wordt gecomplementeerd

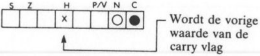
Datastroom:



Timing: 1 M cyclus; 4 T states: 2 usec @ 2 MHz

Adresseringsmode: Impliciet:

Flags:



CP s

Vergelijk operand s met de accumulator

Formaat: A − s

Formaat: s: kan zijn n, (HL), (IX + d), of (IY + d).

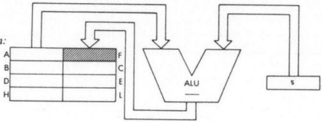
r	1 0 1 1 1 ←r→	
n	1 1 1 1 1 1 1 0	FE
	←n→	byte 2: immediate data
(HL)	1 0 1 1 1 1 1 0	byte 1: BE
(IX + d)	1 1 0 1 1 1 0 1	byte 1: DD
	1 0 1 1 1 1 1 0	byte 2: BE
	←d→	byte 3: verplaatsing
(IY + d)	1 1 1 1 1 1 0 1	byte 1: FD
	1 0 1 1 1 1 1 0	byte 2: BE
	←d→	byte 3: verplaatsing

r kan een van de volgende zijn:

A − 111	E − 011
B − 000	H − 100
C − 001	L − 101
D − 010	

Beschrijving: De gespecificeerde operand wordt van de accumulator afgetrokken. Het resultaat wordt genegeerd. S wordt bij de ADD instructies gedefinieerd

Datastroom:



Timing:

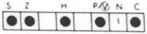
s:	*M cycli:*	*T states:*	*usec @ 2 MHz:*
r	1	4	2
n	2	7	3.5
(HL)	2	7	3.5
(IX + d)	5	19	9.5
(IY + d)	5	19	9.5

Adresseringsmode: r: impliciet; n: onmiddellijk; (HL): indirect; (IX + d), (IY + d): geïndexeerd.

Byte codes: CP r:

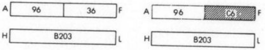
r:	A	B	C	D	E	H	L
	BF	B8	B9	BA	BB	BC	BD

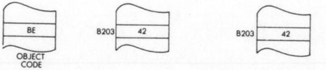
Flags:



Voorbeeld: CP (HL)

Voor: Na:





CPD vergelijk met decrement

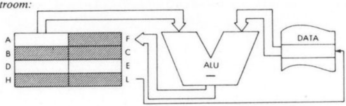
Funktie: A — [HL]; HL ←— HL — 1; BC ←— BC — 1

Formaat:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	0	1	0	0	1	byte 2: A9

Beschrijving: De inhoud van het adres in HL wordt van de inhoud van de accumulator afgetrokken. Het resultaat wordt genegeerd. Daarna worden de inhouden van HL en van BC ieder met 1 verlaagd.

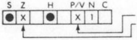
Datastroom:



Timing: 4 M cycli; 16 T states: 8 usec @ 2 MHz

Adresseringsmode: Indirect.

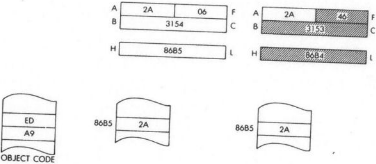
Flags:



Gereset als BC = 0 na instructie, anders geset
Geset als A = [HL]

Voorbeeld: CPD

Voor: Na:



CPDR

Blok vergelijking met decrement

Funktie:

A — [HL]; HL ← HL — 1; BC ← BC — 1;
Herhaal tot BC = 0 of A = [HL]

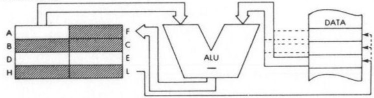
Formaat:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	1	1	0	0	1	byte 2: B9

Beschrijving:

De inhoud van de geheugen locatie, geadresseerd door het HL register wordt van de inhoud van de accumulator afgetrokken. Het resultaat wordt genegeerd. Van zowel BC als HL wordt 1 afgetrokken. Als BC ≠ 0 en A ≠ [HL], dan wordt van de programma teller 2 afgetrokken, en de instructie wordt opnieuw uitgevoerd.

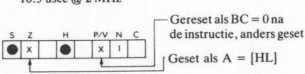
Datastroom:



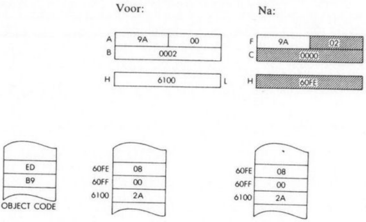
Timing:

BC = 0 of A = [HL]: 4 M cycli; 16 T states:
8 usec @ 2 MHz
BC ≠ 0 en A = [HL]: 5 M cycli; 21 T states:
10.5 usec @ 2 MHz

Flags:



Voorbeeld: CPDR



CPI vergelijk met increment

Funktie: A — [HL]; HL ← HL + 1; BC ← BC — 1

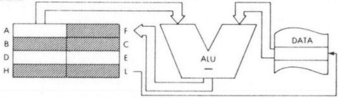
Formaat:

1	1	1	0	1	1	0	1	byte 1: ED

1	0	1	0	0	0	0	1	byte 2: A1

Beschrijving: De inhoud van de geheugen locatie, geadresseerd door HL wordt van de inhoud van de accumulator afgetrokken. Het resultaat wordt genegeerd. Bij de inhoud van HL wordt 1 opgeteld, en van BC wordt 1 afgetrokken.

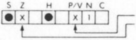
Datastroom:



Timing: 4 M cycli; 16 T states: 8 usec @ 2 MHz

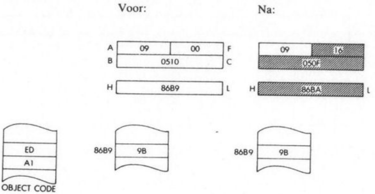
Adresseringsmode: Indirect.

Flags:



Gereset als BC = 0 na de instructie, anders geset
Geset als A = [HL]

Voorbeeld: CPI



CPIR blok vergelijking met increment

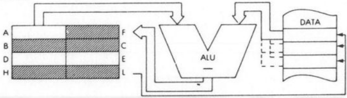
Funktie: A — [HL]; H ← HL + 1; BC ← BC — 1;
Herhaal tot BC = 0 of A = [HL]

Formaat:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	1	0	0	0	1	byte 2: B1

Beschrijving: De inhoud van de geheugen locatie, geadresseerd door HL wordt afgetrokken van de inhoud van de accumulator, en het resultaat wordt genegeerd. Daarna wordt bij HL 1 opgeteld, en van BC wordt 1 afgetrokken. Als BC ongelijk is aan 0, en A ongelijk aan [HL], dan wordt van de programma teller 2 afgetrokken en de instructie wordt opnieuw uitgevoerd.

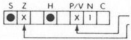
Datastroom:



Timing: BC + L 0 of A = [HL]: 4 M cycli; 16 T states: 8 usec @ 2 MHz
BC ≠ 0 en A ≠ [HL]: 5 M cycli; 21 T states: 10.5 usec @ 2 MHz

Adresseringsmode: Indirect.

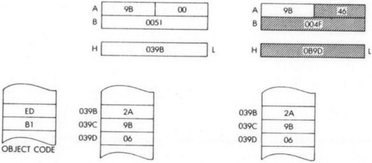
Flags:



Gereset als BC = 0 na de instructie, anders geset.
Geset als A = [HL]

Voorbeeld: CPIR

Voor: Na:



CPL

complementeer de accumulator

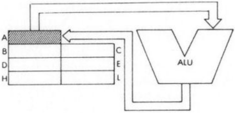
Funktie: $A \leftarrow \overline{A}$

Formaat:

0	0	1	0	1	1	1	1	2F

Beschrijving: De inhoud van de accumulator wordt gecomplementeerd (geinverteerd), en het resultaat gaat naar de accumulator. (een-complement).

Datastroom:



Timing: 1 M cyclus; 4 T states: 2 usec @ 2 MHz

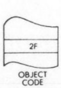
Adresseringsmode: Impliciet.

Flags:

S	Z		H		P/V	N	C
			1			1	

Voorbeeld: CPL

Voor: Na:







DAA maak van de inhoud van de accumulator packed BCD

Funktie: Zie tabel hier onder

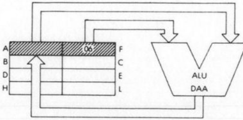
Formaat:

0	0	1	0	0	1	1	1	27

Beschrijving: De instructie telt voorwaardelijk "6" op bij de rechter en/of linker nibble van de accumulator, afhankelijk van het status register, bedoeld voor BCD conversie na een rekenkundige bewerking.

N	C	waarde van hoge nibble	H	waarde van lage nibble	– bij A opgeteld	C na de instructie
0	0	0-9	0	0-9	00	0
(ADD,	0	0-8	0	A-F	06	0
ADC,	0	0-9	1	0-3	06	0
INC)	0	A-F	0	0-9	60	1
	0	9-F	0	A-F	66	1
	0	A-F	1	0-3	66	1
	1	0-2	0	0-9	60	1
	1	0-2	0	A-F	66	1
	1	0-3	1	0-3	66	1
1	0	0-9	0	0-9	00	0
(SUB,	0	0-8	1	6-F	FA	0
SBC,	1	7-F	0	0-9	AO	1
DEC,	1	6-F	1	6-F	9A	1
NEG)						

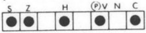
Datastroom:



Timing: 1 M cyclus; 4 T states: 2 usec @ 2 MHz

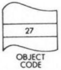
Adresseringsmode: Impliciet:

Flags:



Voorbeeld: DAA

Voor: Na:





DEC m — decrement operand m

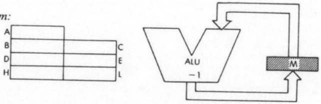
Funktie: m ← m − 1

Formaat: m: kan zijn r, (HL), (IX+d), (IY+d)

r	0 0 ← r → 1 0 1	
(HL)	0 0 1 1 0 1 0 1	35
(IX + d)	1 1 0 1 1 1 0 1	byte 1: DD
	0 0 1 1 0 1 0 1	byte 2: 35
	← d →	byte 3: verplaatsing
(IY + d)	1 1 1 1 1 1 0 1	byte 1: FD
	0 0 1 1 0 1 0 1	byte 2: 35
	← d →	byte 3: verplaatsing

Beschrijving: De inhoud van de geheugen locatie, geadresseerd door de gespecificeerde operand, wordt met 1 verminderd, en weer teruggezet op het zelfde adres. M wordt gedefinieerd bij de INC instructie

Datastroom:



Timing:

m:	M cycli:	T states:	usec @ 2 MHz:
r	1	4	2
(HL)	3	11	5.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Adresseringsmode: r: impliciet; (HL): indirect; (IX + d), (IY + d): geindexeerd.

Byte codes: DEC r

r:	A	B	C	D	E	H	L
	3D	05	0D	15	1D	25	2D

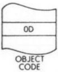
Flags:

S	Z		H		P/V	N	C
●	●		●		●	1	

Voorbeeld: DEC C

Voor: 0F C

Na: 0E C



DEC rr decrement register paar rr

Funktie: rr ← rr − 1

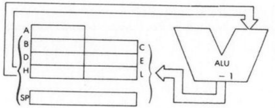
Formaat:

0	0	r	r	1	0	1	1

Beschrijving: De inhoud van het gespecificeerde register paar wordt met 1 verminderd. Het resultaat komt in het zelfde register paar terug. RR kan zijn:

BC - 00	HL - 10
DE - 01	SP - 11

Datastroom:



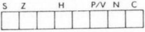
Timing: 1 M cyclus; 6 T states; 3 usec @ 2 MHz

Adresseringsmode: Impliciet.

Byte codes: rr:

BC	DE	HL	SP
0B	1B	2B	3B

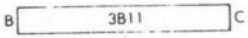
Flags:



(geen invloed).

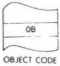
Voorbeeld: DEC BC

Voor:



Na:





DEC IX

decrement IX

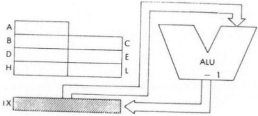
Funktie: IX ← IX − 1

Formaat:

1	1	0	1	1	1	0	1	byte 1: DD
0	0	1	0	1	0	1	1	byte 2: 2B

Beschrijving: Van de inhoud van het IX register wordt 1 afgetrokken, en het resultaat wordt weer in IX geplaatst.

Datastroom:



Timing: 2 M cycli; 10 T states; 5 usec @ 2 MHz

Adresseringsmode: Impliciet.

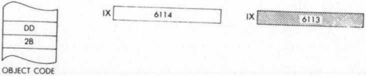
Flags:

S	Z		H		P/V	N	C

(geen invloed).

Voorbeeld: DEC IX

Voor: Na:



DEC IY

decrement IY

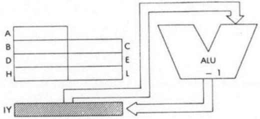
Funktie: IY ← IY − 1

Formaat:

1	1	1	1	1	1	0	1	byte 1: FD
0	0	1	0	1	0	1	1	byte 2: 2B

Beschrijving: Van de inhoud van het register IY wordt 1 afgetrokken, en het resultaat wordt weer in IY geplaatst.

Datastroom:



Timing: 2 M cycli; 10 T states; 5 usec @ 2 MHz

Adresseringsmode: Impliciet:

Flags:

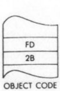
S	Z		H		P/V	N	C

(geen invloed).

Voorbeeld: DEC IY

Voor: Na:





DI interrupts worden niet uitgevoerd

Funktie: IFF ← 0

Formaat:

1	1	1	1	0	0	1	1	F3

Beschrijving: De interrupt flip-flop wordt gereset, waardoor maskeerbare interrupts niet meer worden gehonoreerd. DI mag gebruikt worden tijdens de afhandeling van een interrupt routine, waardoor andere interrupts onmogelijk zijn. Interrupts zijn weer mogelijk na de instructie EI

Timing: 1 M cyclus; 4 T states; 2 usec @ 2 MHz

Adresseringsmode: Impliciet:

Flags:

S	Z		H		P/V	N	C

(geen invloed).

DJNZ e

decrement B en spring e relatief als NZ

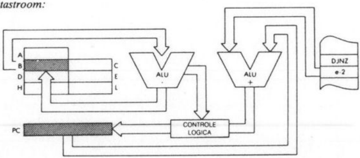
Funktie: $B \leftarrow b - 1$; als $B \neq 0$: $PC \leftarrow PC + e$

Formaat:

0	0	0	1	0	0	0	0	byte 1: 10
e-2								byte 2: verplaatsing

Beschrijving: Van het B register wordt 1 afgetrokken. Als het resultaat niet nul is (NZ), dan wordt de verplaatsing bij de programma teller opgeteld, gebruik makende van het twee-complement. Sprongen voorwaarts en achterwaarts zijn dus mogelijk. De verplaatsing waarde wordt opgeteld bij de waarde van PC + 2 (na de sprong). De effectieve verplaatsing is dus −126 tot +129.

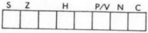
Datastroom:



Timing: $B \neq 0$: 3 M cycli; 13 T states; 6.5 usec @ 2 MHz.
$B = 0$: 2 M cycli; 8 T states; 4 usec @ 2 MHz

Adresseringsmode: Onmiddellijk.

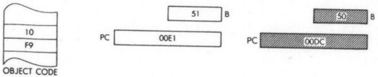
Flags:



(geen invloed).

Voorbeeld: DJNZ $ - 5 ($ = huidige PC)

Voor: Na:



EI processor staat weer interrupts toe

Funktie: IFF ← 1

Formaat:

1	1	1	1	1	0	1	1	FB

Beschrijving: De interrupt flip-flop wordt geset, waardoor gemaskeerde interrupts weer mogelijk zijn na de uitvoering van de instructie volgend op de EI instructie. In de tussentijd worden maskeerbare interrupts niet door de processor gehonoreerd.

Timing: 1 M cyclus; 4 T states; 2 usec @ 2 MHz

Adresseringsmode: Impliciet.

Flags:

S	Z		H		P/V	N	C

(geen invloed).

Voorbeeld: Een gebruikelijke volgorde aan het eind van een interrupt routine is:
EI
RETI
Maskeerbare interrupts zijn dan weer mogelijk na de RETI instructie.

EX AF,AF'

verwissel accumulator en vlaggen met alternatieve registers

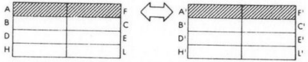
Funktie: AF↔AF'

Formaat:

0	0	0	0	1	0	0	0	08

Beschrijving: De inhouden van de accumulator en status register worden verwisseld met de inhouden van de overeenkomstige registers uit de andere register bank.

Datastroom:



Timing: 1 M cyclus; 4 T states; 2 usec @ 2 MHz

Adresseringsmode: Implicit.

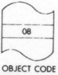
Flags:

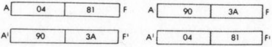
S	Z		H		P/V	N	C

(geen invloed).

Voorbeeld: EX AF, AF'

Voor: Na:





EX DE,HL verwissel HL en DE register

Funktie: DE ⟷ HL

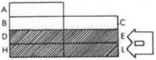
Formaat:

1	1	1	0	1	0	1	1

EB

Beschrijving: De inhouden van DE en HL worden verwisseld

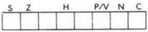
Datastroom:



Timing: 1 M cyclus; 4 T states; 2 usec @ 2 MHz

Adresseringsmode: Impliciet:

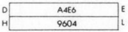
Flags:



(geen invloed).

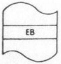
Voorbeeld: EX DE, HL

Voor:



Na:





OBJECT CODE

EX (SP),HL verwissel HL met de top van de stapel

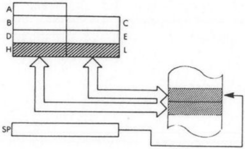
Funktie: (SP) ← L; (SP + 1) ← H

Formaat:

1	1	1	0	0	0	1	1	E3

Beschrijving: De inhoud van het L register wordt verwisseld met de inhoud van het adres waar de stack pointer naar wijst. De inhoud van H wordt verwisseld met de inhoud van het adres volgend op het adres waar de stack pointer naar wijst.

Datastroom:



Timing: 5 M cycli; 19 T states; 9.5 usec @ 2 MHz

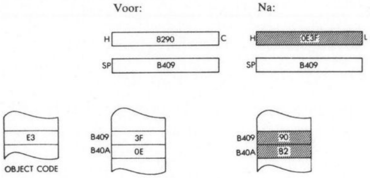
Adresseringsmode: Indirect.

Flags:

S	Z		H		P/V	N	C

(geen invloed).

Voorbeeld: EX (SP), HL



EX (SP),IX — Verwissel IX met de top van de stapel

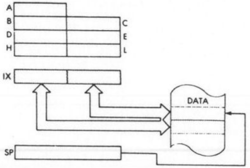
Funktie: (SP) ↔ IX_{laag}; (SP + 1) ↔ IX_{hoog}

Formaat:

1	1	0	1	1	1	0	1	byte 1: DD
1	1	1	0	0	0	1	1	byte 2: E3

Beschrijving: De inhoud van het lage orde deel van IX wordt verwisseld met de inhoud van het geheugen adres waar de stack pointer naar wijst. De inhoud van het hoge orde deel van IX wordt verwisseld met de inhoud van het geheugen adres volgend op het adres waar de stack pointer naar wijst.

Datastroom:



Timing: 6 M cycli; 23 T states; 11.5 usec @ 2 MHz

Adresseringsmode: Indirect.

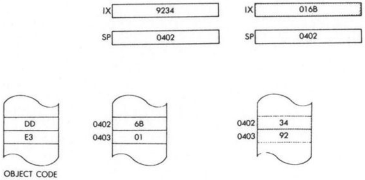
Flags:

S	Z		H		P/V	N	C

(geen invloed).

Voorbeeld: EX (SP), IX

Voor: Na:



EX (SP),IY

Verwissel IY met de top van de stapel

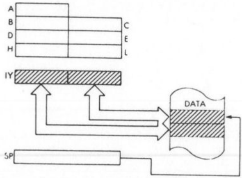
Funktie: $(SP) \leftrightarrow IY_{laag}$; $(SP + 1) \leftrightarrow IY_{hoog}$

Formaat:

1	1	1	1	1	1	0	1	byte 1: FD
1	1	1	0	0	0	1	1	byte 2: E3

Beschrijving: De inhoud van het lage orde deel van IY wordt verwisseld met de inhoud van het geheugen adres waar de stack pointer naar wijst. De inhoud van het hoge orde deel van IY wordt verwisseld met de inhoud van het geheugen adres volgend op het adres waar de stack pointer naar wijst.

Datastroom:



Timing: 6 M cycli; 23 T states; 11.5 usec @ 2 MHz

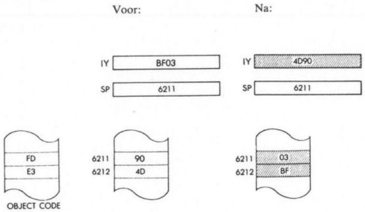
Adresseringsmode: Indirect.

Flags:

S	Z		H		P/V	N	C

(geen invloed).

Voorbeeld: EX (SP), IY



EXX verwissel alternatieve registers

Funktie: BC ↔ BC'; DE ↔ DE'; HL ↔ HL'

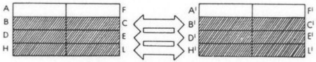
Formaat:

1	1	0	1	1	0	0	1

D9

Beschrijving: De inhouden van de general-purpose registers worden verwisseld met de inhouden van de overeenkomstige registers van de andere geheugen bank.

Datastroom:



Timing: 1 M cyclus; 4 T states; 2 usec @ 2 MHz

Adresseringsmode: Impliciet.

Flags:

S	Z		H		P/V	N	C

(geen invloed).

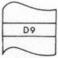
Voorbeeld: EXX

Voor:

A	04	2B	F
B	39	26	C
D	54	02	E
H	F1	D0	L
A'	3F	2A	F'
B'	8C	00	C'
D'	93	D0	E'
H'	4F	E3	L'

Na:

A	04	2B	F
B	8C	00	C
D	93	D0	E
H	4F	E3	L
A'	3F	2A	F'
B'	39	26	C'
D'	54	02	E'
H'	F1	D0	L'



OBJECT CODE

HALT stopt CPU

Funktie: CPU werking stilgelegd.

Formaat:

0	1	1	1	0	1	1	0	76

Beschrijving: De CPU stopt het programma, en voert alleen NOP's uit voor de refresh van het geheugen, totdat een interrupt of een reset wordt ontvangen.

Timing: 1 M cyclus; 4 T states; 2 usec @ 2 MHz + onbepaald aantal Nop's.

Adresseringsmode: Impliciet.

Flags:

S	Z		H		P/V	N	C

(geen invloed).

IM 0 — Kies interrupt mode 0

Funktie: Interne interrupt controle

Formaat:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	0	0	0	1	1	0	byte 2: 46

Beschrijving: Kies interrupt mode 0. In deze toestand kan het apparaat, dat de interrupt uitvoert, een instructie op de data bus zetten, die de processor uit moet voeren. De eerste byte van de instructie moet gedurende de interrupt acknowledge cyclus verschijnen.

Timing: 2 M cycli; 8 T states; 4 usec @ 2 MHz

Adresseringsmode: Impliciet.

Flags:

S	Z		H		P/V	N	C

(geen invloed).

IM 1 kies interrupt mode 1

Funktie: Interne interrupt controle

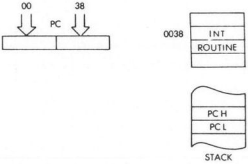
Formaat:

1	1	1	0	1	1	0	1	byte 1: ED

0	1	0	1	0	1	1	0	byte 2: 56

Beschrijving: Kies interrupt mode 1. Er wordt een RST 0038H instructie uitgevoerd, als de interrupt verschijnt.

Datastroom:



Timing: 2 M cycli; 8 T states; 4 usec @ 2 MHz

Adresseringsmode: Impliciet.

Flags:

S	Z		H		P/V	N	C

(geen invloed).

IM 2 kies interrupt mode 2

Funktie: Interne interrupt controle

Formaat:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	0	1	1	1	1	0	byte 2: 5E

Beschrijving: Kies interrupt mode 2. Bij het optreden van de interrupt, moet het apparaat, dat de interrupt veroorzaakt, een data byte op de data bus zetten. Dat byte vormt het lage orde deel van het adres. Het hoge orde deel van het adres is afkomstig van het I register. Dit adres wijst naar een tweede adres in het geheugen, waarmee de programma teller geladen wordt.

Timing: 2 M cycli; 8 T states; 4 usec @ 2 MHz

Adresseringsmode: Impliciet.

Flags:

S	Z		H		P/V	N	C	
								(geen invloed).

IN r,(C) — laad register vanuit poort (C)

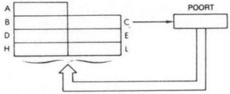
Funktie: r ← (C)

Formaat:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	←	r	→	0	0	0	byte 2

Beschrijving: Het randapparaat, geadresseerd door de inhoud van het C register, wordt gelezen, en het resultaat komt in het gespecificeerde register.
C vormt de bits A0 tot en met A7 van de adres bus.
B vormt de bits A8 tot en met A15.

Datastroom:



r kan een van de volgende zijn:

A – 111
B – 000
C – 001
D – 010
E – 011
H – 100
L – 101

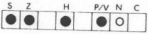
Timing: 3 M cycli; 12 T states; 6 usec @ 2 MHz

Adresseringsmode: Extern.

Byte codes:

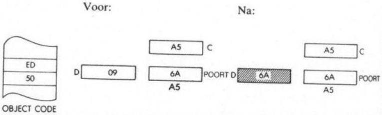
r:	A	B	C	D	E	H	L
ED	78	40	48	50	58	60	68

Flags:



Het is belangrijk te weten, dat de instructie IN A,(N) geen invloed heeft op de vlaggen, terwijl IN r,(C) dat wel heeft.

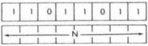
Voorbeeld: IN D, (C)



IN A,(N) laad accumulator vanuit input poort N

Funktie: A ← (N)

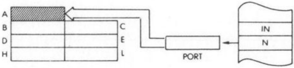
Formaat:



byte 1: DB
byte 2: poort adres

Beschrijving: Randapparaat N wordt gelezen, en het resultaat komt in de accumulator.
De waarde N wordt op bits A0 tot en met A7 van de adres bus geplaatst. A vormt bits A8 tot en met A15.

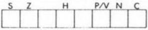
Datastroom:



Timing: 3 M cycli; 11 T states; 5.5 usec @ 2 MHz

Adresseringsmode: Extern.

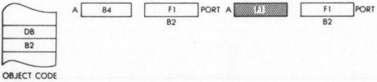
Flags:



(geen invloed).

Voorbeeld: IN A, (B2)

Voor: Na:



INC r — increment register r

Funktie: r ← r + 1

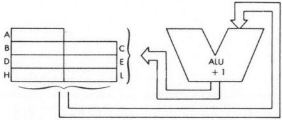
Formaat:

0	0	← r →	1	0	0

Beschrijving: Bij de inhoud van het gespecificeerde register wordt 1 opgeteld. R kan zijn:

A - 111	E - 011
B - 000	H - 100
C - 001	L - 101
D - 010	

Datastroom:



Timing: 1 M cyclus; 4 T states; 2 usec @ 2 MHz

Adresseringsmode: Impliciet.

Byte codes:

r:	A	B	C	D	E	H	L
	3C	04	0C	14	1C	24	2C

Flags:

S	Z		H		P/V	N	C
●	●		●		●	○	

Voorbeeld: INC D



Voor: D 06

Na: D 07

INC rr — increment register paar rr

Funktie: rr ← rr + 1

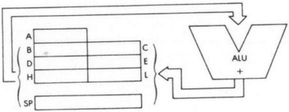
Formaat:

0	0	r	r	0	0	1	1

Beschrijving: Bij de inhoud van het gespecificeerde register paar wordt 1 opgeteld. Het resultaat komt in het zelfde register paar. RR kan zijn:

BC - 00	HL - 10
DE - 01	SP - 11

Datastroom:



Timing: 1 M cyclus; 6 T states; 3 usec @ 2 MHz

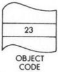
Adresseringsmode: Impliciet.

Byte codes:

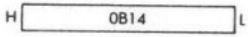
rr:	BC	DE	HL	SP
	03	13	23	33

Flags: S Z H P/V N C (geen invloed).

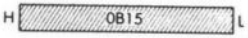
Voorbeeld: INC HL



Voor:



Na:



INC (HL) incrementeer de indirect geadresseerde geheugen locatie (HL)

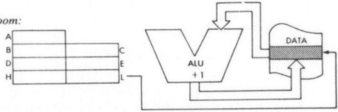
Funktie: (HL) ← (HL) + 1

Formaat:

0	0	1	1	0	1	0	0	34

Beschrijving: De inhoud van de geheugen locatie, geadresserd door het register paar HL, wordt met 1 verhoogd. Het resultaat komt op het zelfde adres terug.

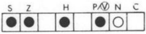
Datastroom:



Timing: 3 M cycli; 11 T states; 5.5 usec @ 2 MHz

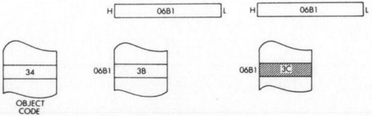
Adresseringsmode: Indirect.

Flags:



Voorbeeld: INC (HL)

Voor: Na:



INC (IX + d) incrementeer de geindexeerd geadresseerde geheugen locatie (IX + d)

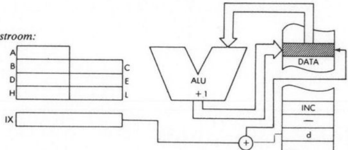
Funktie: (IX + d) ← (IX + d) + 1

Formaat:

1	1	0	1	1	1	0	1	byte 1: DD
0	0	1	1	0	1	0	0	byte 2: 34
d								byte 3: verplaatsing

Beschrijving: De inhoud van de geheugen locatie, geadresseerd door de inhoud van het index register IX plus de gegeven verplaatsing, wordt met 1 verhoogd, en wordt geladen op het zelfde adres.

Datastroom:



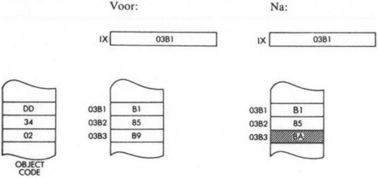
Timing: 6 M cycli; 23 T states; 11.5 usec @ 2 MHz

Adresseringsmode: Geïndexeerd:

Flags:

S	Z		H		P/V	N	C
●	●		●		●	○	

Voorbeeld: INC (IX + 2)



INC (IY + d) incrementeer de geindexeerd geadresseerde geheugen locatie (IY + d)

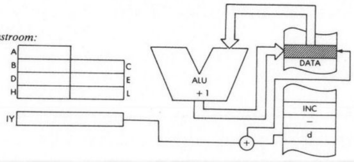
Funktie: (IY + d) ← (IY + d) + 1

Formaat:

1	1	1	1	1	1	0	1	byte 1: FD
0	0	1	1	0	1	0	0	byte 2: 34
←			d				→	byte 3: verplaatsing

Beschrijving: De inhoud van de geheugen locatie, geadresseerd door de inhoud van het index register IY plus de gegeven verplaatsing, wordt met 1 verhoogd, en wordt geladen op het zelfde adres.

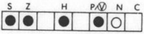
Datastroom:



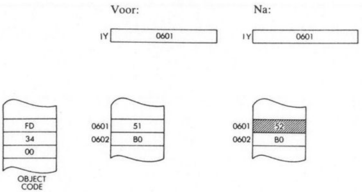
Timing: 6 M cycli; 23 T states; 11.5 usec @ 2 MHz

Adresseringsmode: Geïndexeerd.

Flags:



Voorbeeld: INC (IY + 0)



INC IX

incrementeer IX

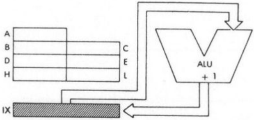
Funktie: IX ← IX + 1

Formaat:

1	1	0	1	1	1	0	1	byte 1: DD
0	0	1	0	0	0	1	1	byte 2: 23

Beschrijving: De inhoud van het register IX wordt met 1 verhoogd, en terug geplaatst in IX.

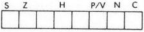
Datastroom:



Timing: 2 M cycli; 10 T states; 5 usec @ 2 MHz

Adresseringsmode: Impliciet.

Flags:



(geen invloed).

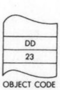
Voorbeeld: INC IX

Voor:



Na:





INC IY — incrementeer IY

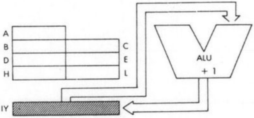
Funktie: IY ← IY + 1

Formaat:

1	1	1	1	1	1	0	1	byte 1: FD
0	0	1	0	0	0	1	1	byte 2: 23

Beschrijving: De inhoud van het register IY wordt met 1 verhoogd, en terug geplaatst in IY.

Datastroom:



Timing: 2 M cycli; 10 T states; 5 usec @ 2 MHz

Adresseringsmode: Impliciet.

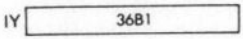
Flags:

S	Z		H		P/V	N	C

(geen invloed).

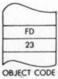
Voorbeeld: INC IY

Voor:



Na:





IND input met decrement

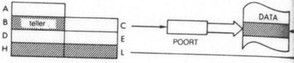
Funktie: (HL) ← (C); B ← B − 1; HL ← HL − 1

Formaat:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	0	1	0	1	0	byte 2: AA

Beschrijving: Het randapparaat, geadresseerd door het register C, wordt gelezen, en het resultaat wordt geladen op het adres waar HL naar wijst. Van zowel het B register, als het register paar HL wordt daarna 1 afgetrokken.

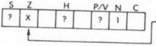
Datastroom:



Timing: 4 M cycli; 16 T states; 8 usec @ 2 MHz

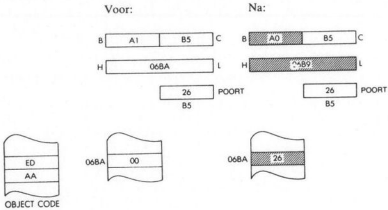
Adresseringsmode: Extern.

Flags:



Geset als B = 0 na de instructie, anders gereset.

Voorbeeld: IND



INDR blok input met decrement

Funktie:

(HL) ← (C); B ← B − 1; HL ← HL − 1
Herhaal tot B = 0

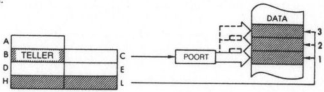
Formaat:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	1	1	0	1	0	byte 2: BA

Beschrijving:

Het randapparaat, geadresseerd door het C register, wordt gelezen, en het resultaat wordt geladen op het adres waar HL naar wijst. Daarna wordt van B en HL 1 afgetrokken. Als B niet nul is, wordt van de programma teller 2 afgetrokken, en de instructie wordt opnieuw uitgevoerd.

Datastroom:



Timing:

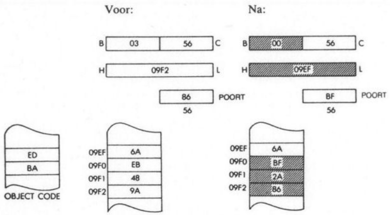
B = 0:4 M cycli; 16 T states; 8 usec @ 2 MHz.
B ≠ 0:5 M cycli; 21 T states; 10.5 usec @ 2 MHz.

Adresseringsmode: Extern

Flags:

S	Z		H		P/V	N	C
?	1		?		?	1	

Voorbeeld: INDR



INI

input met increment

Funktie: (HL) ← (C); B ← B − 1; HL ← HL + 1

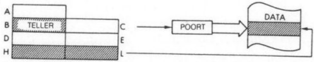
Formaat:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	0	0	0	1	0	byte 2: A2

Beschrijving: Het randapparaat, geadresseerd door register C, wordt gelezen, en het resultaat wordt geladen op het adres waar HL naar wijst. Van het B register wordt 1 afgetrokken, en bij HL wordt 1 opgeteld.

De inhoud van C vormt de lage orde helft van de adres bus. De inhoud van B wordt op de hoge orde helft van de adres bus geplaatst. I/O wordt gewoonlijk geselecteerd door C (A0 t/m A7). B is een byte teller.

Datastroom:



Timing: 4 M cycli; 16 T states; 8 usec @ 2 MHz

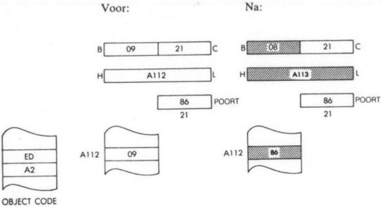
Adresseringsmode: Extern.

Flags:

S	Z		H		P/V	N	C
?	X		?		?	1	

Z wordt geset, als na de instructie geldt: B = 0, anders wordt Z gereset.

Voorbeeld: INI



INIR — blok input met increment

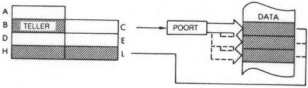
Funktie: (HL) ← (C); B ← B − 1; HL ← HL + 1; Herhaal tot B = 0

Formaat:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	1	0	0	1	0	byte 2: B2

Beschrijving: Het door C geadresseerde randapparaat wordt gelezen, en het resultaat wordt geladen op de geheugen locatie, geadresseerd door HL. Van B wordt 1 afgetrokken, en bij HL wordt 1 opgeteld. Als B niet nul is, dan wordt de programma teller met 2 verlaagd, en de instructie wordt opnieuw uitgevoerd.

Datastroom:



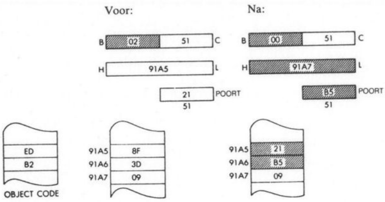
Timing: B = 0:4 M cycli; 16 T states; 8 usec @ 2 MHz.
B ≠ 0:5 M cycli; 21 T states; 10.5 usec @ 2 MHz.

Adresseringsmode: Extern.

Flags:

S	Z		H		P/V	N	C
?	1		?		?	1	

Voorbeeld: INIR

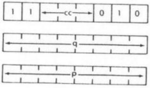


JP cc,pq

spring op voorwaarde naar adres pq

Funktie: als cc waar is: PC ← pq

Formaat:



byte 1

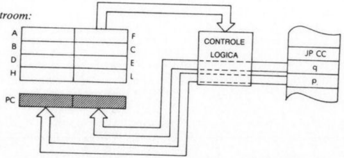
byte 2: adres, laag

byte 3: adres, hoog

Beschrijving: Indien voldaan is aan de gespecificeerde voorwaarde, dan wordt het twee-bytes adres volgend op de opcode, geladen in de programma teller. Het eerste byte vormt het lage orde deel van PC en het tweede byte het hoge orde deel. Indien de voorwaarde niet waar is, dan wordt het adres genegeerd. CC kan zijn:

NZ - 000	niet nul
Z - 001	nul
NC - 010	geen carry
C - 011	carry
PO - 100	pariteit oneven
PE - 101	pariteit even
P - 110	plus
M - 111	minus

Datastroom:



Timing: 3 M cycli; 10 T states; 5 usec @ 2 MHz

Adresseringsmode: Onmiddellijk.

Byte codes: C C

NZ	Z	NC	C	PO	PE	P	M
C2	CA	D2	DA	E2	EA	F2	FA

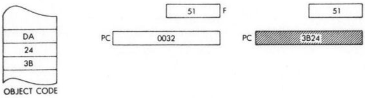
Flags:

S	Z		H		P/V	N	C

(geen invloed).

Voorbeeld: JP C, 3B24

Voor: Na:

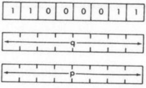


JP pq

spring naar adres pq

Funktie: PC ← pq

Formaat:



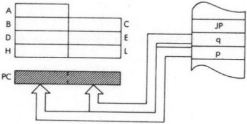
byte 1: C3

byte 2: adres, laag

byte 3: adres, hoog

Beschrijving: De inhoud van het geheugen adres volgend op de opcode wordt in het lage orde deel van PC geladen, de inhoud van het tweede adres na de opcode komt in het hoge orde deel van de programma teller. De volgende instructie wordt van dat adres gehaald.

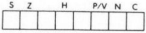
Datastroom:



Timing: 3 M cycli; 10 T states, 5 usec @ 2 MHz

Adresseringsmode: Onmiddellijk.

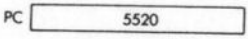
Flags:



(geen invloed).

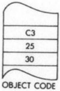
Voorbeeld: JP 3025

Voor:



Na:





JP (HL) spring naar HL

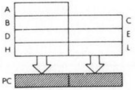
Funktie: PC ← HL

Formaat:

1	1	1	0	1	0	0	1	E9

Beschrijving: De inhoud van het register paar HL wordt in de programma teller geladen. De volgende instructie wordt van dit adres gehaald.

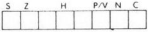
Datastroom:



Timing: 1 M cyclus; 4 T states; 2 usec @ 2 MHz

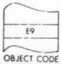
Adresseringsmode: Impliciet:

Flags:

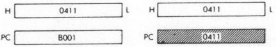


(geen invloed).

Voorbeeld: JP (HL)



Voor: Na:



JP (IX) spring naar IX

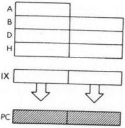
Funktie: PC ← IX

Formaat:

1	1	0	1	1	1	0	1	byte 1: DD
1	1	1	0	1	0	0	1	byte 2: E9

Datastroom: De inhoud van het register IX wordt in de programma teller geladen. De volgende instructie wordt van dit adres gehaald.

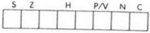
Datastroom:



Timing: 2 M cycli; 8 T states; 4 usec @ 2 MHz

Adresseringsmode: Impliciet:

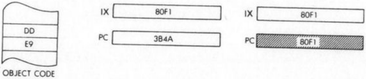
Flags:



(geen invloed).

Voorbeeld: JP (IX)

Voor: Na:



JP (IY) spring naar IY

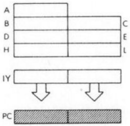
Funktie: PC ← IY

Formaat:

1	1	1	1	1	1	0	1	byte 1: FD
1	1	1	0	1	0	0	1	byte 2: E9

Beschrijving: De inhoud van het register IY wordt in de programma teller geladen. De volgende instructie wordt van dit adres gehaald.

Datastroom:



Timing: 2 M cycli; 8 T states; 4 usec @ 2 MHz

Adresseringsmode: Impliciet.

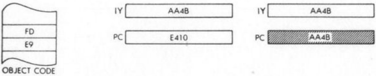
Flags:

S	Z		H		P/V	N	C

(geen invloed).

Voorbeeld: JP (IY)

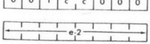
Voor: Na:



JR cc,e spring voorwaardelijk e relatief

Funktie: als cc waar is: PC ← PC + e

Formaat:



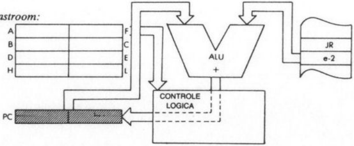
byte 1

byte 2: Verplaatsing

Beschrijving: Als voldaan wordt aan de gespecificeerde voorwaarde, dan wordt de gegeven verplaatsing opgeteld bij de programma teller, gebruik makende van de twee-complements rekenkunde. Voor- en achterwaartse sprongen zijn dus mogelijk. De verplaatsing wordt opgeteld bij PC + 2. De effectieve verplaatsing ligt dus tussen −126 en +129 bytes. De vertaler (assembler) trekt automatisch 2 af van de oorspronkelijke verplaatsing om de hex code te genereren. Als niet aan de voorwaarde wordt voldaan, dan wordt de verplaatsing genegeerd, en wordt de volgende instructie uitgevoerd. CC kan zijn:

NZ - 00	NC - 10
Z - 01	C - 11

Datastroom:



Timing:

	M cycli:	*T states:*	*usec* @ 2 MHz:
voorwaarde waar	3	12	6
Voorwaarde niet waar	2	7	3.5

Adresseringsmode: Onmiddellijk.

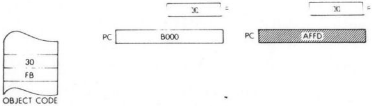
Byte codes:

cc:	NZ	Z	NC	C
	20	28	30	38

Flags: S Z H P/V N C (geen invloed).

Voorbeeld: JR NC, $ – 3 $ = huidige PC

Voor: Na:



JR e Spring e relatief

Funktie: PC ← PC + e

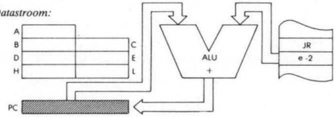
Formaat:

0	0	0	1	1	0	0	0	byte 1: 18

← e-2 → byte 2: Verplaatsing

Beschrijving: De gegeven verplaatsing wordt bij de programma teller opgeteld, gebruik makende van de tweecomplements rekenkunde, zodat voor- en achterwaartse sprongen mogelijk zijn. De verplaatsing wordt bij PC + 2 (na de instructie) opgeteld. Effectief bereik: −126 tot +129.

Datastroom:



Timing: 3 M cycli; 12 T states; 6 usec @ 2 MHz

Adresseringsmode: Onmiddellijk.

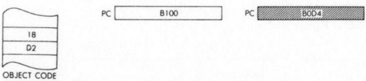
Flags:

S	Z		H		P/V	N	C

(geen invloed).

Voorbeeld: JR D4

Voor: Na:



LD dd,(nn) — laad register paar dd vanuit het geheugen adres in

Funktie: $dd_{laag} \leftarrow (nn); dd_{hoog} \leftarrow (nn + 1)$

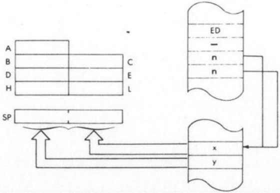
Formaat:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	d	d	1	0	1	1	byte 2
← n →								byte 3: adres, laag
← n →								byte 4: adres, hoog

Beschrijving: De inhoud van het adres, waar naar de twee bytes na de opcode wijzen, wordt in het lage orde deel van het gespecificeerde register paar geladen. De inhoud van het daarop volgende adres wordt in het hoge orde deel geladen. Het lage orde deel van het nn adres volgt direct op de opcode. DD kan zijn:

BC - 00	HL - 10
DE - 01	SP - 11

Datastroom:



Timing: 6 M cycli; 20 T states; 10 usec @ 2 MHz

Adresseringsmode: : Direct.

Byte codes:

dd: BC	DE	HL	SP
4B	5B	6B	7B

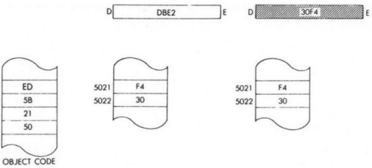
Flags:

S	Z		H		P/V	N	C

(geen invloed).

Voorbeeld: LD DE, (5021)

Voor: Na:



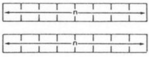
LD dd,nn

laad register paar dd met de data nn

Funktie: dd ← nn

Formaat:

0	0	d	d	0	0	0	1	byte 1



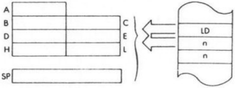
byte 2: onmiddellijk data, laag

byte 3: onmiddellijk data, hoog

Beschrijving: De inhoud van de adressen volgend op de opcode, wordt in het gespecificeerde register paar geladen. Het lage orde deel van de data staat direct na de opcode. DD kan zijn:

BC - 00	HL - 10
DE - 01	SP - 11

Datastroom:



Timing: 3 M cycli; 10 T states; 5 usec @ 2 MHz

Adresseringsmode: Onmiddellijk.

Byte codes:

dd:	BC	DE	HL	SP
	01	11	21	31

Flags:

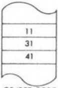
S	Z		H		P/V	N	C

(geen invloed).

Voorbeeld: LD DE, 4131

Voor: Na:





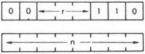
OBJECT CODE

LD R,n

laad register R met data n

Funktie: r ← n

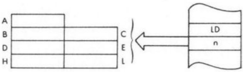
Formaat:



Beschrijving: De inhoud van het adres volgend op de opcode wordt in het gespecificeerde register geladen. R kan zijn:

A - 111	E - 011
B - 000	H - 100
C - 001	L - 101
D - 010	

Datastroom:



Timing: 2 M cycli; 7 T states; 3.5 usec @ 2 MHz

Adresseringsmode: Onmiddellijk.

Byte codes:

r:	A	B	C	D	E	H	L
	3E	06	0E	16	1E	26	2E

Flags:

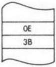
S	Z		H		P/V	N	C

(geen invloed).

Voorbeeld: LD C, 3B

Voor: Na:





OBJECT CODE

LD r,r' laad register r met r'

Funktie: r ← r'

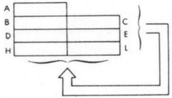
Formaat:

0	1	← r →	← r' →

Beschrijving: De inhoud van het register r' wordt in het register r geladen. R en r' kunnen zijn:

A - 111
B - 000
C - 001
D - 010
E - 011
H - 100
L - 101

Datastroom:



Timing: 1 M cyclus; 4 T states; 2 usec @ 2 MHz

Adresseringsmode: Impliciet.

Byte codes:

(bestemming) \ (bron)	A	B	C	D	E	H	L
A	7F	78	79	7A	7B	7C	7D
B	47	40	41	42	43	44	45
C	4F	48	49	4A	4B	4C	4D
D	57	50	51	52	53	54	55
E	4F	58	59	5A	5B	5C	5D
H	67	60	61	62	63	64	65
L	6F	68	69	6A	6B	6C	6D

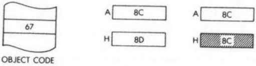
Flags:

S	Z		H		P/V	N	C

(geen invloed).

Voorbeeld: LD H, A

Voor: Na:



LD (BC),A

schrijf de accumulator naar het indirect geadresseerde adres (BC)

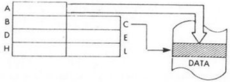
Funktie: (BC) ← A

Formaat:

0	0	0	0	0	0	1	0	02

Beschrijving: De accumulator wordt geschreven in de geheugen locatie, die geadresseerd wordt door de inhoud van het register paar BC

Datastroom:



Timing: 2 M cycli; 7 T states; 3.5 usec @ 2 MHz

Adresseringsmode: Indirect.

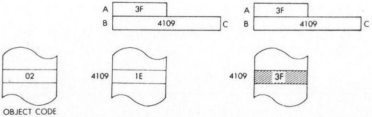
Flags:

S	Z		H		P/V	N	C

(geen invloed).

Voorbeeld: LD (BC), A

Voor: Na:



LD (DE),A schrijf de accumulator naar het indirect geadresseerde adres (DE)

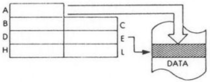
Funktie: (DE) ← A

Formaat:

0	0	0	1	0	0	1	0	12

Beschrijving: De accumulator wordt geschreven in de geheugen locatie, die geadresseerd wordt door de inhoud van het registerpaar DE

Datastroom:



Timing: 2 M cycli; 7 T states; 3.5 usec @ 2 MHz

Adresseringsmode: Indirect.

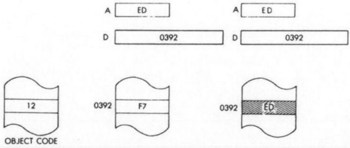
Flags:

S	Z		H		P/V	N	C

(geen invloed).

Voorbeeld: LD (DE), A

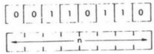
Voor: Na:



LD (HL),n schrijf data n naar het adres, waar HL naar wijst

Funktie: (HL) ← n

Formaat:

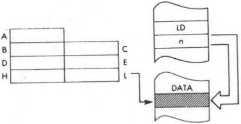


byte 1: 36

byte 2: onmiddellijk data

Beschrijving: De inhoud van het adres volgend op de opcode wordt geladen in de geheugen locatie, waar HL (data teller) naar wijst

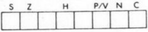
Datastroom:



Timing: 3 M cycli; 10 T states; 5 usec @ 2 MHz

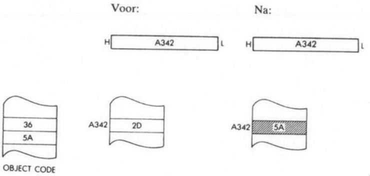
Adresseringsmode: Onmiddellijk/indirect.

Flags:



(geen invloed).

Voorbeeld: LD (HL), 5A



LD (HL),r beschrijf de indirect geadresseerde geheugen locatie (HL) met de inhoud van register r

Funktie: (HL) ← r

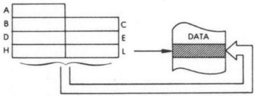
Formaat:

0	1	1	1	0	← r →

Beschrijving: De inhoud van het gespecificeerde register wordt geladen in de geheugen locatie, die geadresseerd wordt door het register paar HL. R kan zijn:

A - 111	E - 011
B - 000	H - 100
C - 001	L - 101
D - 010	

Datastroom:



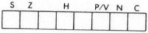
Timing: 2 M cycli; 7 T states; 3.5 usec @ 2 MHz

Adresseringsmode: Indirect.

Byte codes:

r:	A	B	C	D	E	H	L
	77	70	71	72	73	74	75

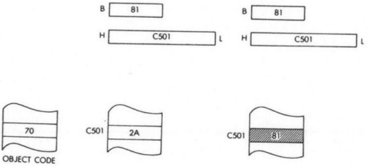
Flags:



(geen invloed).

Voorbeeld: LD (HL), B

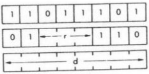
Voor: Na:



LD r,(IX + d) laad register r met de inhoud van de geïndexeerd geadresseerde locatie (IX + d)

Funktie: r ← (IX + d)

Formaat:



byte 1: DD

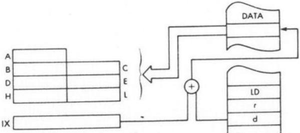
byte 2

byte 3: Verplaatsing

Beschrijving: De inhoud van de locatie geadresseerd door het IX register plus de verplaatsing, wordt in het gespecificeerde register geladen. R kan zijn:

A - 111	E - 011
B - 000	H - 100
C - 001	L - 101
D - 010	

Datastroom:



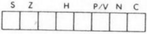
Timing: 5 M cycli; 19 T states; 9.5 usec @ 2 MHz

Adresseringsmode: Geïndexeerd:

Byte codes:

r:	A	B	C	D	E	H	L	
DD-	7E	46	4E	56	5E	66	6E	-d

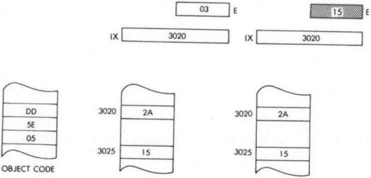
Flags:



(geen invloed).

Voorbeeld: LD E, (IX + 5)

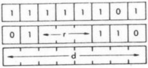
Voor: Na:



LD r,(IY + d) laad register r met de inhoud van de geindexeerd geadresseerde locatie (IY + d)

Funktie: r ← (IY + d)

Formaat:

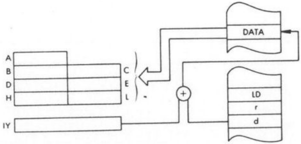


byte 1: FD
byte 2
byte 3: Verplaatsing

Beschrijving: De inhoud van de locatie geadresseerd door het IY register plus de verplaatsing, wordt in het gespecificeerde register geladen. R kan zijn:

A - 111
B - 000
C - 001
D - 010
E - 011
H - 100
L - 101

Datastroom:



Timing: 5 M cycli; 19 T states; 9.5 usec @ 2 MHz

Adresseringsmode: Geïndexeerd:

Byte codes:

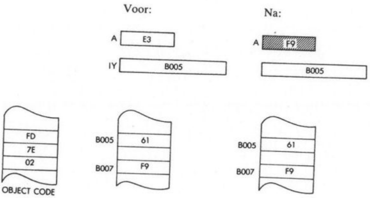
r:	A	B	C	D	E	H	L	
FD-	7E	46	4E	56	56	66	6E	- d

Flags:

S	Z		H		P/V	N	C

(geen invloed).

Voorbeeld: LD A, (IY + 2)



LD (IX + d),n laad de geindexeerd geadresseerde geheugen locatie (IX + d) met data n

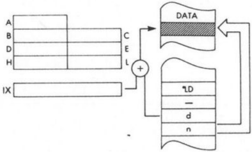
Funktie: (IX + d) ← n

Formaat:

1	1	0	1	1	1	0	1	byte 1: DD
0	0	1	1	0	1	1	0	byte 2: 36
← d →								byte 3: verplaatsing
← n →								byte 4: onmiddellijk data

Beschrijving: De inhoud van het geheugen adres volgend op de opcode wordt geladen in de geheugen locatie, die geadresseerd wordt door de inhoud van het index register plus de gegeven verplaatsing

Datastroom:



Timing: 5 M cycli; 19 T states; 9.5 usec @ 2 MHz

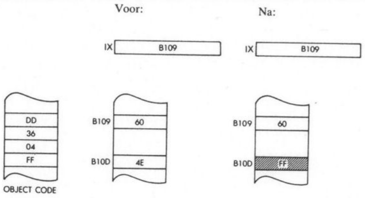
Adresseringsmode: Geïndexeerd: Impliciet:

Flags:

S	Z		H		P/V	N	C

(geen invloed).

Voorbeeld: LD (IX + 4), FF



LD (IY + d),n laad de geindexeerd geadresseerde geheugen locatie (IY + d) met data n

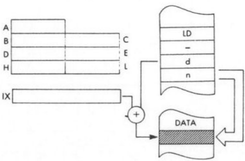
Funktie: (IY + d) ← n

Formaat:

1	1	1	1	1	1	0	1	byte 1: FD
0	0	1	1	0	1	1	0	byte 2: 36
←			d				→	byte 3: Verplaatsing
←			n				→	byte 4: onmiddellijk data

Beschrijving: De inhoud van het geheugen adres volgend op de opcode wordt geladen in de geheugen locatie, die geadresseerd wordt door de inhoud van het index register plus de gegeven verplaatsing

Datastroom:



Timing: 5 M cycli; 19 T states; 9.5 usec @ 2 MHz

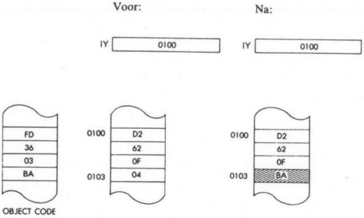
Adresseringsmode: Impliciet: Geïndexeerd:

Flags:

S	Z		H		P/V	N	C

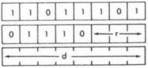
(geen invloed).

Voorbeeld: LD (IY + 3), BA



LD (IX + d),r laad de inhoud van register r in de geindexeerd geadresseerde geheugen locatie (IX + d)

Formaat:



byte 1: DD

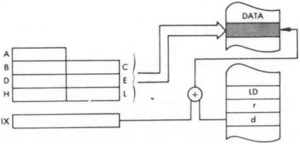
byte 2

byte 3: Verplaatsing

Beschrijving: De inhoud van het gespecificeerde register wordt geladen op de geheugen locatie, geadresseerd door de inhoud van het index register plus de gegeven verplaatsing. R kan zijn:

A - 111	E - 011
B - 000	H - 100
C - 001	L - 101
D - 010	

Datastroom:



Timing: 5 M cycli; 19 T states; 9.5 usec @ 2 MHz

Adresseringsmode: Geïndexeerd.

Byte codes:

r:	A	B	C	D	E	H	L	
DD-	77	70	71	72	73	74	75	- d

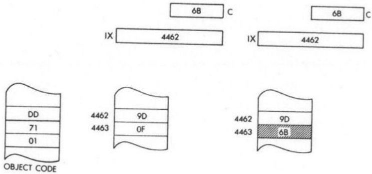
Flags:

S	Z		H		P/V	N	C

(geen invloed).

Voorbeeld: LD (IX + 1), C

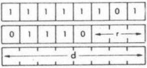
Voor: Na:



LD (IY + d),r laad de inhoud van register r in de geindexeerd geadresseerde geheugen locatie (IY + d)

Funktie: (IY + d) ← r

Formaat:



byte 1: FD

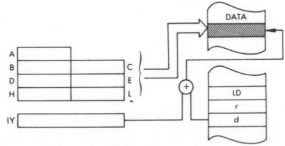
byte 2

byte 3: Verplaatsing

Beschrijving: De inhoud van het gespecificeerde register wordt geladen op de geheugen locatie, geadresseerd door de inhoud van het index register plus de gegeven verplaatsing. R kan zijn:

A - 111	E - 011
B - 000	H - 100
C - 001	L - 101
D - 010	

Datastroom:



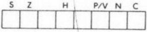
Timing: 5 M cycli; 19 T states; 9.5 usec @ 2 MHz

Adresseringsmode: Geïndexeerd.

Byte codes:

r:	A	B	C	D	E	H	L	
FD-	77	70	71	72	73	74	75	-d

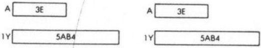
Flags:

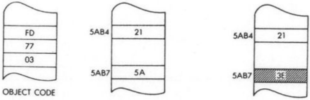


(geen invloed).

Voorbeeld: LD (IY + 3), A

Voor: Na:

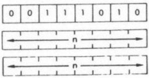




LD A,(nn) laad de accumulator met de inhoud van de geheugen locatie (nn)

Funktie: A ← (nn)

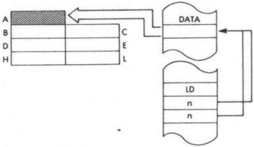
Formaat:



byte 1: 3A
byte 2: adres, laag
byte 3: adres, hoog

Beschrijving: De inhoud van de geheugen locatie, geadresseerd door de twee bytes volgend op de opcode, wordt in de accumulator geladen. Het lage orde byte van het adres staat meteen na de opcode.

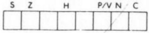
Datastroom:



Timing: 4 M cycli; 13 T states; 6.5 usec @ 2 MHz

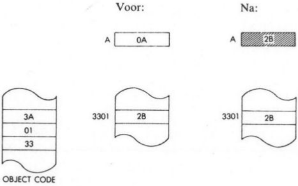
Adresseringsmode: Direct.

Flags:



(geen invloed).

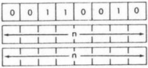
Voorbeeld: LD A, (3301)



LD (nn),A — laad de accumulator op de direct geadresseerde geheugen locatie (nn)

Funktie: (nn) ← A

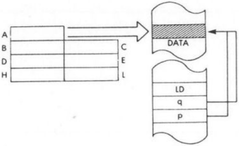
Formaat:



byte 1: 32
byte 2: adres, laag
byte 3: adres, hoog

Beschrijving: De inhoud van de accumulator worden geladen in de geheugen locatie, geadresseerd door de twee bytes volgend op de opcode. Het lage orde deel van het adres staat direct achter de opcode.

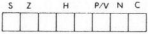
Datastroom:



Timing: 4 M cycli; 13 T states; 6.5 usec @ 2 MHz

Adresseringsmode: Direct.

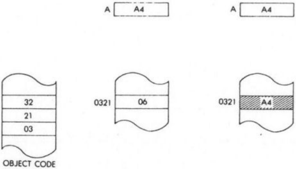
Flags:



(geen invloed).

Voorbeeld: LD (0321), A

Voor: Na:



LD (nn),dd laad register paar dd op de geheugen locatie, geadresseerd door nn

Funktie: (nn) ← dd_{laag}, (nn + 1) ← dd_{hoog}

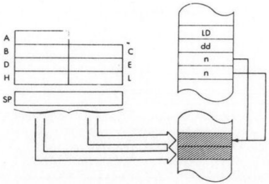
Formaat:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	d	d	0	0	1	1	byte 2
←			n				→	byte 3: adres, laag
←			n				→	byte 4: adres, hoog

Beschrijving: De inhoud van het lage orde deel van het gespecificeerde adres wordt geladen in de geheugen locatie, geadresseerd door de twee bytes volgend op de opcode. In het daarop volgende adres wordt de inhoud van het hoge orde deel van het register paar geladen. Het lage orde deel van het adres volgt meteen op de opcode. DD kan zijn:

BC - 00	HL - 10
DE - 01	SP - 11

Datastroom:



Timing: 6 M cycli; 20 T states; 10 usec @ 2 MHz

Adresseringsmode: Direct.

Byte codes:

dd:	BC	DE	HL	SP
ED-	43	53	63	73

Flags:

S	Z		H		P/V	N	C

(geen invloed).

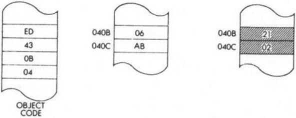
Voorbeeld: LD (040B), BC

Voor:

B 0221 C

Na:

B 0221 C



LD (nn),HL

schrijf register paar HL naar geheugen locatie (nn)

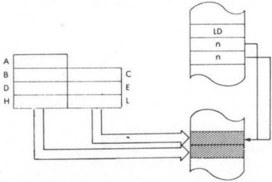
Funktie: (nn) ← L; (nn + 1) ← H

Formaat:

0	0	1	0	0	0	1	0	byte 1: 22
←			n				→	byte 2: adres, laag
←			n				→	byte 3: adres, hoog.

Beschrijving: De inhoud van register L wordt geladen op de geheugen locatie, geadresseerd door de twee bytes volgend op de opcode. Het daarop volgende adres wordt geladen met de inhoud van het H register. Het lage orde deel van adres nn staat direct achter de opcode.

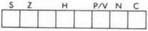
Datastroom:



Timing: 5 M cycli; 16 T states; 8 usec @ 2 MHz

Adresseringsmode: Direct.

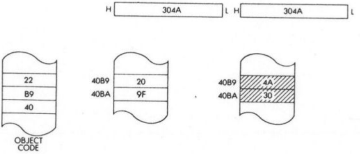
Flags:



(geen invloed).

Voorbeeld: LD (40B9), HL

Voor: Na:



LD (nn),IX laad IX op geheugen locatie, geadresseerd door nn

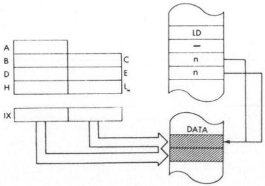
Funktie: $(nn) \leftarrow IX_{laag}$; $(nn + 1) \leftarrow IX_{hoog}$

Formaat:

1	1	0	1	1	1	0	1	byte 1: DD
0	0	1	0	0	0	1	0	byte 2: 22
n								byte 3: adres, laag
n								byte 4: adres, hoog

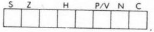
Beschrijving: De inhoud van het lage orde deel van register IX wordt geladen op de geheugen locatie, die geadresseerd wordt door de twee bytes volgend op de opcode. De inhoud van het hoge orde deel van IX wordt geladen op het adres, volgend op het adres, waar het lage orde deel geladen is. Het lage orde deel van het adres staat direct achter de opcode.

Datastroom:



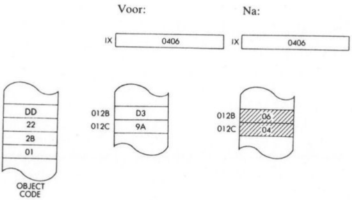
Timing: 6 M cycli; 20 T states: 10 usec @ 2 MHz

Adresseringsmode: Direct.

Flags: 

(geen invloed).

Voorbeeld: LD (012B), IX



LD (nn),IY

schrijf IY naar geheugen locatie, geadresseerd door nn

Funktie: $(nn) \leftarrow IY_{laag}$, $(nn + 1) \leftarrow IY_{hoog}$

Formaat:

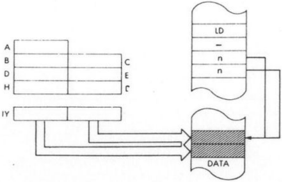
1	1	1	1	1	1	0	1	byte 1: FD
0	0	1	0	0	0	1	0	byte 2: 22

← n → byte 3: adres, laag

← n → byte 4: adres, hoog

Beschrijving: De inhoud van het lage orde deel van register IY wordt geladen op de geheugen locatie, die geadresseerd wordt door de twee bytes volgend op de opcode. De inhoud van het hoge orde deel van IY wordt geladen op het adres, volgend op het adres, waar het lage orde deel geladen is. Het lage orde deel van het adres staat direct achter de opcode.

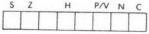
Datastroom:



Timing: 6 M cycli; 20 T states; 10 usec @ 2 MHz

Adresseringsmode: Direct.

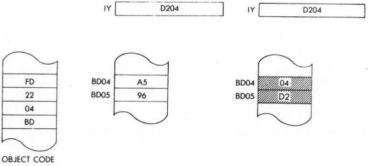
Flags:



(geen invloed).

Voorbeeld: **LD (BD04), IY**

Voor: Na:



LD A,(BC)

laad de accumulator vanuit het indirect door BC geadresseerde geheugenadres.

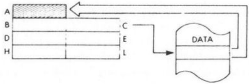
Funktie: A ← (BC)

Formaat:

0	0	0	0	1	0	1	0	0A

Beschrijving: De inhoud van de geheugen locatie, geadresseerd door de inhoud van het BC register paar, wordt in de accumulator geladen.

Datastroom:



Timing: 2 M cycli; 7 T states; 3.5 usec @ 2 MHz

Adresseringsmode: Indirect.

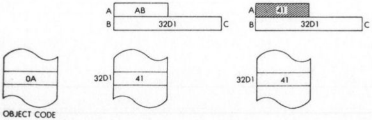
Flags:

S	Z		H		P/V	N	C

(geen invloed).

Voorbeeld: LD A, (BC)

Voor: Na:



LD A,(DE) laad de accumulator vanuit het indirect door DE geadresseerde geheugenadres.

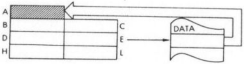
Funktie: A ← (DE)

Formaat:

0	0	0	1	1	0	1	0	1A

Beschrijving: De inhoud van de geheugen locatie, geadresseerd door de inhoud van het DE register paar, wordt in de accumulator geladen.

Datastroom:



Timing: 2 M cycli; 7 T states; 3.5 usec @ 2 MHz

Adresseringsmode: Indirect.

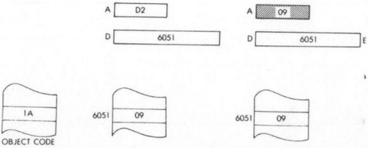
Flags:

S	Z		H		P/V	N	C

(geen invloed).

Voorbeeld: LD A, (DE)

Voor: Na:



LD A,I

laad het interrupt vector register I in de accumulator

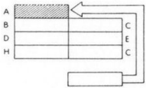
Funktie: A ← I

Formaat:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	0	1	0	1	1	1	byte 2: 57

Beschrijving: De inhoud van het interrupt vector register wordt in de accumulator geladen

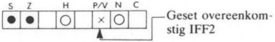
Datastroom:



Timing: 2 M cycli; 9 T states; 4.5 usec @ 2 MHz

Adresseringsmode: Impliciet.

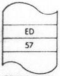
Flags:



Voorbeeld: LD A, I

Voor: Na:





OBJECT CODE

LD I,A

laad de accumulator in het interrupt vector register I

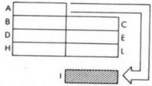
Funktie: I ← A

Formaat:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	0	0	0	1	1	1	byte 2: 47

Beschrijving: De inhoud van de accumulator wordt in het interrupt vector register I geladen.

Datastroom:



Timing: 2 M cycli; 9 T states; 4.5 usec @ 2 MHz

Adresseringsmode: Impliciet.

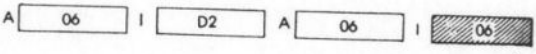
Flags:

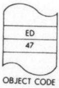
S	Z		H		P/V	N	C

(geen invloed).

Voorbeeld: LD I, A

Voor: Na:





LD A,R

laad het geheugen refresh register in de accumulator

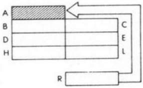
Funktie: A ← R

Formaat:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	0	1	1	1	1	1	byte 2: 5F

Beschrijving: De inhoud van het geheugen refresh register wordt in de accumulator geladen.

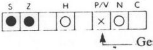
Datastroom:



Timing: 2 M cycli; 9 T states; 4.5 usec @ 2 MHz

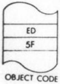
Adresseringsmode: Impliciet.

Flags:



Voorbeeld: LD A, R

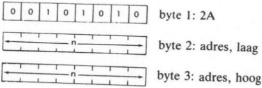




LD HL,(nn) laad het HL register vanuit het door nn geadresseerde geheugen adres

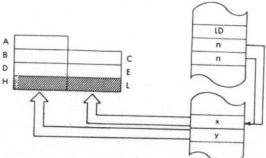
Funktie: L ← (nn); H ← (nn + 1)

Formaat:



Beschrijving: De inhoud van het geheugen adres, geadresseerd door de twee bytes volgend op de opcode, wordt in register L geladen. De inhoud van het adres, volgend op het adres, waarvan de inhoud in L geladen is, wordt in register H geladen. Het lage orde byte van het adres staat direct achter de opcode.

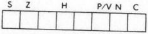
Datastroom:



Timing: 5 M cycli; 16 T states; 8 usec @ 2 MHz

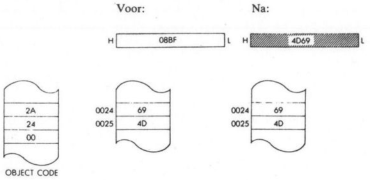
Adresseringsmode: Direct.

Flags:



(geen invloed).

Voorbeeld: LD HL, (0024)



LD IX,nn

laad het register IX met de immediate data nn

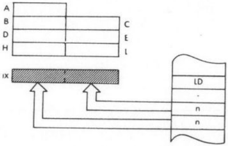
Funktie: IX ← nn

Formaat:

1	1	0	1	1	1	0	1	byte 1: DD
0	0	1	0	0	0	0	1	byte 2: 21
←			n				→	byte 3: onmiddellijk data laag
←			n				→	byte 4: onmiddellijk data hoog

Beschrijving: De inhouden van de twee bytes, die direct na de opcode staan, worden in het register IX geladen.

Datastroom:



Timing: 4 M cycli; 14 T states; 7 usec @ 2 MHz

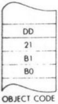
Adresseringsmode: Onmiddellijk.

Flags:

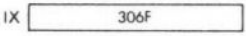
S	Z		H		P/V	N	C

(geen invloed).

Voorbeeld: LD IX, BOB1



Voor:



Na:



LD IX,(nn) laad de door nn geadresseerde geheugen plaats in het register IX

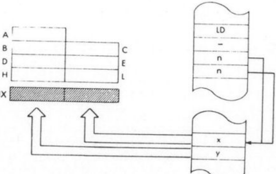
Funktie: $IX_{laag} \leftarrow (nn)$; $IX_{hoog} \leftarrow (nn + 1)$

Formaat:

1	1	0	1	1	1	0	1	byte 1: DD
0	0	1	0	1	0	1	0	byte 2: 2A
← n →								byte 3: adres, laag
← n →								byte 4: adres, hoog

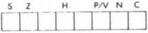
Beschrijving: De inhoud van de geheugen locatie, geadresseerd door de eerste twee bytes na de opcode, wordt in het lage orde deel van IX geladen. De inhoud van de daar op volgende geheugen locatie wordt in het hoge orde deel van IX geladen. Het lage orde deel van het adres staat direct achter de opcode.

Datastroom:



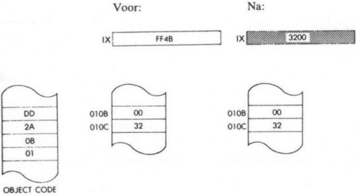
Timing: 6 M cycli; 20 T states; 10 usec @ 2 MHz

Adresseringsmode: Direct.

Flags: 

(geen invloed).

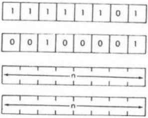
Voorbeeld: LD IX, (010B)



LD IY,nn laad het register IY met de immediate data nn

Funktie: IY ← nn

Formaat:



byte 1: FD

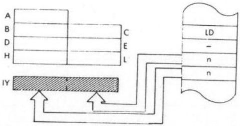
byte 2: 21

byte 3: onmiddellijk data laag

byte 4: onmiddellijk data hoog

Beschrijving: De inhouden van de twee bytes, die direct na de opcode staan, worden in het register IY geladen.

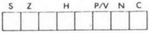
Datastroom:



Timing: 4 M cycli; 14 T states; 7 usec @ 2 MHz

Adresseringsmode: Onmiddellijk.

Flags:

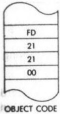


(geen invloed).

Voorbeeld: LD IY, 21

Voor: Na:





LD IY,(nn)

laad de door nn geadresseerde geheugen plaats in het register IY

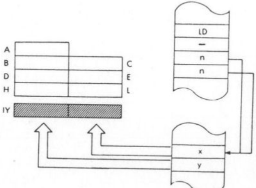
Funktie: $IY_{laag} \leftarrow (nn); IY_{hoog} \leftarrow (nn + 1)$

Formaat:

1	1	1	1	1	1	0	1	byte 1: FD
0	0	1	0	1	0	1	0	byte 2: 2A
←			n				→	byte 3: adres, laag
←			n				→	byte 4: adres, hoog

Beschrijving: De inhoud van de geheugen locatie, geadresseerd door de eerste twee bytes na de opcode, wordt in het lage orde deel van IY geladen. De inhoud van de daar op volgende geheugen locatie wordt in het hoge orde deel van IY geladen. Het lage orde deel van het adres staat direct achter de opcode.

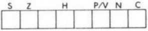
Datastroom:



Timing: 6 M cycli; 20 T states; 10 usec @ 2 MHz

Adresseringsmode: Direct.

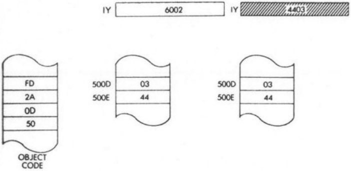
Flags:



(geen invloed).

Voorbeeld: LD IY, (500D)

Voor: Na:



LD R,A

laad de accumulator in het geheugen refresh register R

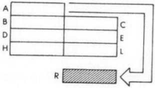
Funktie: R ← A

Formaat:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	0	0	1	1	1	1	byte 2: 4F

Beschrijving: De inhoud van de accumulator wordt in het geheugen refresh register geladen.

Datastroom:



Timing: 2 M cycli; 9 T states; 4.5 usec @ 2 MHz

Adresseringsmode: Impliciet.

Flags:

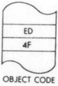
S	Z		H		P/V	N	C

(geen invloed).

Voorbeeld: LD R, A

Voor: A 0F R 40

Na: A 0F R 0F



LD SP,HL

laad HL in de stack pointer

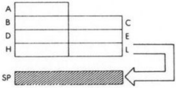
Funktie: SP ← HL

Formaat:

1	1	1	1	1	0	0	1	F9

Beschrijving: De inhoud van het HL register wordt in de stack pointer geladen.

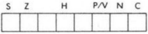
Datastroom:



Timing: 1 M cyclus; 6 T states; 3 usec @ 2 MHz

Adresseringsmode: Impliciet.

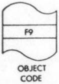
Flags:

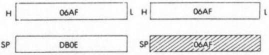


(geen invloed).

Voorbeeld: LD SP, HL

Voor: Na:





LD SP,IX laad het register IX in de stack pointer

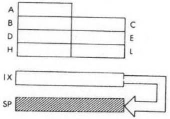
Funktie: SP ← IX

Formaat:

1	1	0	1	1	1	0	1	byte 1: DD
1	1	1	1	1	0	0	1	byte 2: F9

Beschrijving: De inhoud van het register IX wordt in de stack pointer geladen

Datastroom:



Timing: 2 M cycli; 10 T states; 5 usec @ 2 MHz

Adresseringsmode: Impliciet.

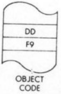
Flags:

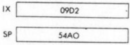
S	Z		H		P/V	N	C

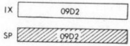
(geen invloed).

Voorbeeld: LD SP, IX

Voor: Na:







LD SP,IY

laad het IY register in de stack pointer

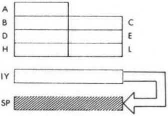
Funktie: SP ← IY

Formaat:

1	1	1	1	1	1	0	1	byte 1: FD
1	1	1	1	1	0	0	1	byte 2: F9

Beschrijving: De inhoud van het register IY worden in de stack pointer geladen.

Datastroom:



Timing: 2 M cycli; 10 T states; 5 usec @ 2 MHz

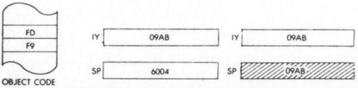
Adresseringsmode: Impliciet.

Flags:

S	Z		H		P/V	N	C	
								(geen invloed).

Voorbeeld: LD SP, IY

Voor: Na:



LDD

blok laden met decrement

Funktie:

(DE) ← (HL); DE ← DE − 1; HL ← HL − 1;
BC ← BC − 1

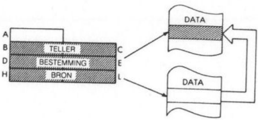
Formaat:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	0	1	0	0	0	byte 2: A8

Beschrijving:

De inhoud van de geheugen plaats, geadresseerd door HL wordt geladen in de geheugen plaats, geadresseerd door DE. Daarna wordt van BC, DE en van HL ieder 1 afgetrokken.

Datastroom:



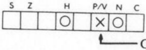
Timing:

4 M cycli; 16 T states; 8 usec @ 2 MHz

Adresseringsmode:

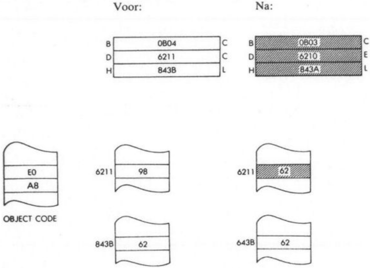
Indirect.

Flags:



Gereset als BC = 0 na de instructie, anders geset.

Voorbeeld: LDD



LDDR herhaald blok laden met decrement

Funktie:

(DE) ← (HL); DE ← DE − 1; HL ← HL − 1;
BC ← BC − 1; Herhaal tot BC = 0

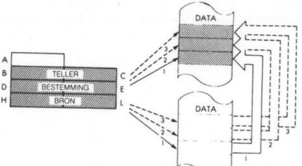
Formaat:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	1	1	0	0	0	byte 2: B8

Beschrijving:

De inhoud van de geheugen locatie, geadresseerd door HL, wordt in de geheugen locatie, geadresseerd door DE, geladen. Daarna wordt van DE, HL en BC 1 afgetrokken. Als BC ongelijk 0 is, dan wordt de programma teller met 2 verminderd, en de instructie wordt opnieuw uitgevoerd.

Datastroom:



Timing:

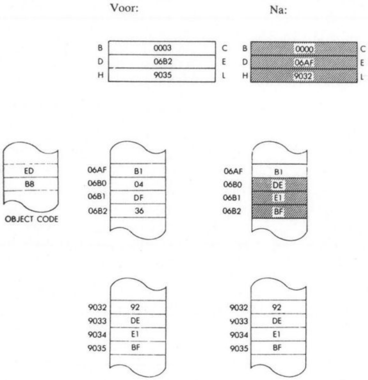
BC ≠ 0: 5 M cycli; 21 T states; 10.5 usec @ 2 MHz.
BC = 0: 4 M cycli; 16 T states; 8 usec @ 2 MHz

Adresseringsmode: Indirect.

Flags:

S	Z		H		P/V	N	C
			O		O	O	

Voorbeeld: LDDR



LDI

blok laden met increment

Funktie:

(DE) ← (HL); DE ← DE + 1; HL ← HL + 1;
BC ← BC − 1

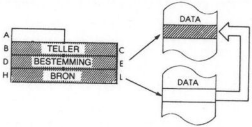
Formaat:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	0	0	0	0	0	byte 2: AO

Beschrijving:

De inhoud van de geheugen plaats, geadresseerd door HL wordt geladen in de geheugen plaats, geadresseerd door DE. Daarna wordt bij DE en HL ieder 1 opgeteld. Van BC wordt 1 afgetrokken.

Datastroom:



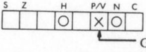
Timing:

4 M cycli; 16 T states; 8 usec @ 2 MHz

Adresseringsmode:

Indirect.

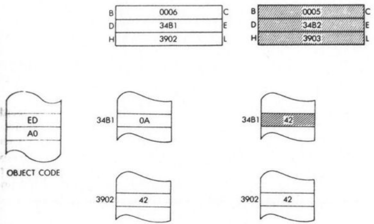
Flags:



Gereset als BC = 0 na de instructie, anders geset.

Voorbeeld: LDI

Voor: Na:



LDIR herhaald blok laden met increment

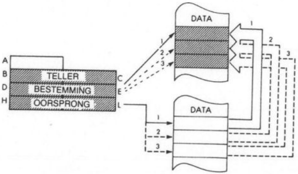
Funktie: (DE) ← (HL); DE ← DE + 1; HL ← HL + 1; BC ← BC − 1; Herhaal tot BC = 0

Formaat:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	1	0	0	0	0	byte 2: BO

Beschrijving: De inhoud van de geheugen locatie, geadresseerd door HL, wordt in de geheugen locatie, geadresseerd door DE, geladen. Daarna wordt bij DE en HL 1 opgeteld. Van BC wordt 1 afgetrokken. Als BC ongelijk 0 is, dan wordt de programma teller met 2 verminderd, en de instructie wordt opnieuw uitgevoerd.

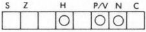
Datastroom:



Timing: For BC ≠ 0: 5 M cycli; 21 T states; 10.5 usec @ 2 MHz.
For BC = 0: 4 M cycli; 16 T states; 8 usec @ 2 MHz

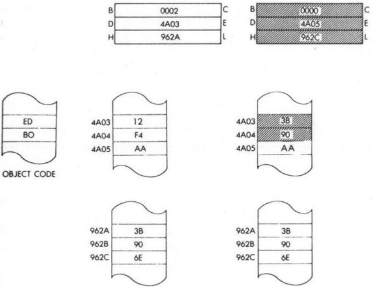
Adresseringsmode: Indirect.

Flags:



Voorbeeld: LDIR

Voor: Na:



LD r,(HL) laad de inhoud van geheugen adres (HL) in register r

Funktie: r ← (HL)

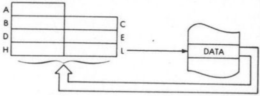
Formaat:

0	1	←	r	→	1	1	0

Beschrijving: De inhoud van de geheugen plaats, geadresseerd door HL, wordt in het gespecificeerde register geladen. R kan zijn:

A - 111	E - 011
B - 000	H - 100
C - 001	L - 101
D - 010	

Datastroom:



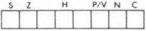
Timing: 2 M cycli; 7 T states; 3.5 usec @ 2 MHz

Adresseringsmode: Indirect.

Byte codes:

r:	A	B	C	D	E	H	L
	7E	46	4E	56	5E	66	6E

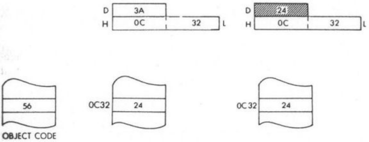
Flags:



(geen invloed).

Voorbeeld: LD D, (HL)

Voor: Na:



NEG trek de accumulator af van 0

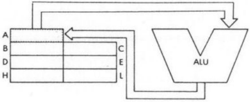
Funktie: A ← 0 − A

Formaat:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	0	0	0	1	0	0	byte 2: 44

Beschrijving: De inhoud van de accumulator wordt van 0 afgetrokken (twee-complement), en het resultaat wordt weer in de accumulator geladen.

Datastroom:



Timing: 2 M cycli; 8 T states; 4 usec @ 2 MHz

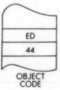
Adresseringsmode: Impliciet.

Flags:

S	Z		H		P/V	N	C
●	●		●		●	1	●

C wordt geset als A 0 was voor de instructie
P wordt geset als A 80H was

Voorbeeld: NEG



Voor: A 32

Na: A CE

NOP geen operatie

Funktie: vertraging

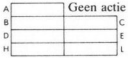
Formaat:

0	0	0	0	0	0	0	0

00

Beschrijving: Gedurende 1 M cyclus wordt niets gedaan.

Datastroom:



Timing: 1 M cyclus; 4 T states; 2 usec @ 2 MHz

Adresseringsmode: Impliciet:

Flags:

S	Z		H		P/V	N	C

(geen invloed).

OR s logische OF van de accumulator en operand s

Funktie: A ← A V s

Formaat: s: kan zijn r, n, (HL), (IX + d), of (IY + d)

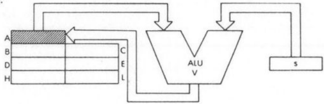
r	1	0	1	1	0	←	r	→	
n	1	1	1	1	0	1	1	0	byte 1: F6
	←			n				→	byte 2: onmiddellijk data
(HL)	1	0	1	1	0	1	1	0	byte 1: B6
(IX + d)	1	1	0	1	1	1	0	1	byte 1: DD
	1	0	1	1	0	1	1	0	byte 2: B6
	←			d				→	byte 3: verplaatsing
(IY + d)	1	1	1	1	1	1	0	1	byte 1: FD
	1	0	1	1	0	1	1	0	byte 2: B6
	←			d				→	byte 3: verplaatsing

r kan zijn:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Beschrijving: De accumulator en de gespecificeerde operand worden logisch ge"of"t. Het resultaat komt in de accumulator. S wordt in de ADD instructie gedefinieerd.

Datastroom:



Timing:

s:	M cycli:	T states:	usec @ 2 MHz:
r	1	4	4
n	2	7	3.5
(HL)	2	7	3.5
(IX + d)	5	19	9.5
(IY + d)	5	19	9.5

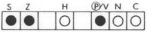
Adresseringsmode: r: impliciet; n: onmiddellijk; (HL): indirect; (IX + d), (IY + d): geïndexeerd.

Byte codes: OR r

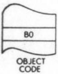
r:

A	B	C	D	E	H	L
B7	B0	B1	B2	B3	B4	B5

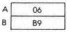
Flags:



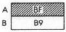
Voorbeeld: OR B



Voor:



Na:



OTDR

blok output met decrement

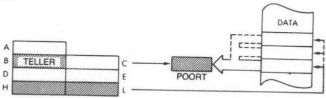
Funktie: (C)←(HL); B B - 1; HL←HL - 1; Herhaal tot B = 0.

Formaat:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	1	1	0	1	1	byte 2: BB

Beschrijving: De inhoud van de geheugen locatie, geadresseerd door HL, wordt op de output poort gezet, die door register C wordt geadresseerd. Van B en HL wordt daarna 1 afgetrokken. Als B ongelijk is aan 0, dan wordt van de programma teller 1 afgetrokken, en de instructie wordt opnieuw uitgevoerd. C vormt bits A0 t/m A7 op de adres bus. De bits A8 t/m A15 komen van B (na de aftrekking)

Datastroom:



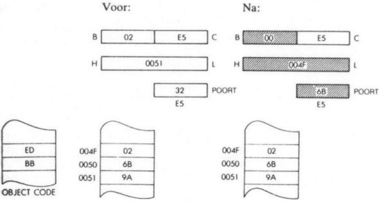
Timing: B = 0: 4 M cycli; 16 T states; 8 usec @ 2 MHz.
B ≠ 0: 5 M cycli; 21 T states; 10.5 usec @ 2 MHz.

Adresseringsmode: Extern.

Flags:

S	Z		H		P/V	N	C
?	1		?		?	1	

Voorbeeld: OTDR



OTIR blok output met increment

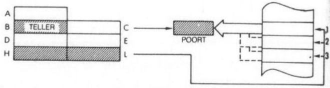
Funktie: (C)←(HL); B←B − 1; HL←HL + 1; Herhaal tot B = 0

Formaat:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	1	0	0	1	1	byte 2: B3

Beschrijving: De inhoud van de geheugen locatie, geadresseerd door HL, wordt op de output poort gezet, die door register C wordt geadresseerd. Van B wordt daarna 1 afgetrokken, en bij HL wordt 1 opgeteld. Als B ongelijk is aan 0, dan wordt van de programma teller 1 afgetrokken, en de instructie wordt opnieuw uitgevoerd. C vormt bits A0 t/m A7 op de adres bus. De bits A8 t/m A15 komen van B (na de aftrekking)

Datastroom:



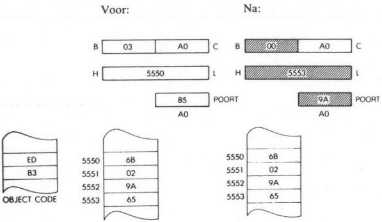
Timing: B = 0: 4 M cycli; 16 T states; 8 usec @ 2 MHz.
B ≠ 0: 5 M cycli; 21 T states; 10.5 usec @ 2 MHz.

Adresseringsmode: Extern.

Flags:

S	Z		H		P/V	N	C
?	1		?		?	1	

Voorbeeld: OTIR



OUT (C),r — zet register r op outputpoort C

Funktie: (C) ← r

Formaat:

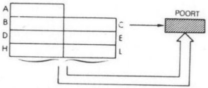
1	1	1	0	1	1	0	1	byte 1: ED
0	1	←	r	→	0	0	1	byte 2

Beschrijving: De inhoud van register r wordt op de door register C geadresseerde outputpoort gezet. R kan zijn:

A - 111	E - 011
B - 000	H - 100
C - 001	L - 101
D - 010	

Register C vormt bits A0 t/m A7 van de adres bus, register B de bits A8 t/m A15.

Datastroom:



Timing: 3 M cycli; 12 T states; 6 usec @ 2 MHz.

Adresseringsmode: Extern.

Flags:

S	Z		H		P/V	N	C

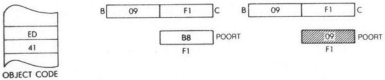
(geen invloed).

Byte codes:

r:	A	B	C	D	E	H	L
	79	41	49	51	59	61	69

Voorbeeld: OUT (C), B

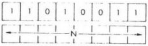
Voor: Na:



OUT (N),A zet de accumulator op outputpoort N

Funktie: (N) ← A

Formaat:

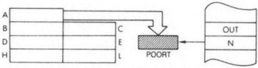


byte 1: D3

byte 2: poort adres

Beschrijving: De inhoud van de accumulator wordt op de output poort, die door de inhoud van het op de opcode volgende byte geadresseerd wordt, gezet.

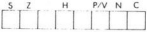
Datastroom:



Timing: 3 M cycli; 11 T states; 5.5 usec @ 2 MHz

Adresseringsmode: Extern.

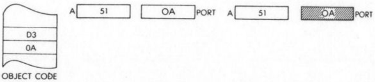
Flags:



(geen invloed).

Voorbeeld: OUT (0A), A

Voor: Na:



OUTD output met decrement

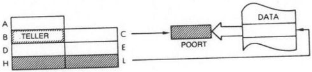
Funktie: (C) ← (HL); BC ← B − 1; HL ← HL − 1

Formaat:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	0	1	0	1	1	byte 2: AB

Beschrijving: De inhoud van de geheugen locatie, geadresseerd door het HL register paar, wordt op de output poort, geadresseerd door de inhoud van het register C, gezet. Daarna worden zowel B als HL met 1 verlaagd. C vormt bits A0 t/m A7 van de adres bus, en B (na de aftrekking) bits A8 t/m A15.

Datastroom:



Timing: 4 M cycli; 16 T states; 8 usec @ 2 MHz

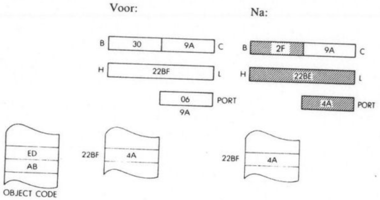
Adresseringsmode: Extern.

Flags:

S	Z		H		P/V	N	C
?	X		?		?	1	

Geset als B = 0 na de instructie, anders gereset.

Voorbeeld: OUTD



OUTI output met increment

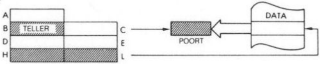
Funktie: (C) ← (HL); B ← B − 1; HL ← HL + 1

Formaat:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	0	0	0	1	1	byte 2: A3

Beschrijving: De inhoud van de geheugen locatie, geadresseerd door het HL register paar, wordt op de output poort, geadresseerd door de inhoud van het register C, gezet. B wordt met 1 verlaagd. Bij HL wordt 1 opgeteld. C vormt bits A0 t/m A7 van de adres bus, en B (na de aftrekking) bits A8 t/m A15.

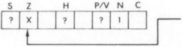
Datastroom:



Timing: 4 M cycli; 16 T states; 8 usec @ 2 MHz

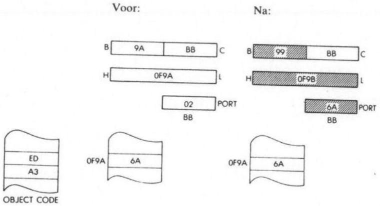
Adresseringsmode: Extern.

Flags:



Geset als B = 0 na de instructie, anders gereset.

Voorbeeld: OUTI



POP qq

haal register paar qq van de stapel

Funktie: $qq_{laag} \leftarrow (SP); qq_{hoog} \leftarrow (SP+1); SP \leftarrow SP+2$

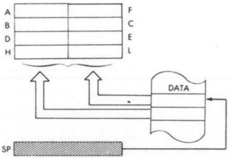
Formaat:

1	1	q	q	0	0	0	1

Beschrijving: De inhoud van de geheugen locatie, geadresseerd door de stackpointer, wordt geladen in het lage orde deel van het gespecificeerde register paar. De stackpointer wordt met 1 verhoogd. De inhoud van het adres waar de stackpointer nu naar wijst wordt geladen in het hoge orde deel van het gespecificeerde register paar. De stackpointer wordt met 1 verhoogd. QQ kan zijn:

BC - 00	HL - 10
DE - 01	AF - 11

Datastroom:



Timing: 3 M cycli; 10 T states; 5 usec @ 2 MHz

Adresseringsmode: Indirect.

Byte codes:

qq:	BC	DE	HL	AF
	C1	D1	E1	F1

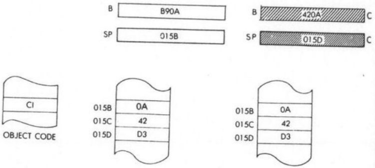
Flags:

S	Z		H		P/V	N	C

(geen invloed).

Voorbeeld: POP BC

Voor: Na:



POP IX

plaats de top van de stapel in IX

Funktie:

$IX_{laag} \leftarrow (SP); IX_{hoog} \leftarrow (SP+1); SP \leftarrow SP+2$

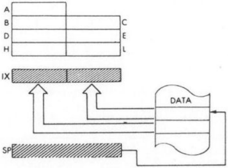
Formaat:

1	1	0	1	1	1	0	1	byte 1: DD
1	1	1	0	0	0	0	1	byte 2: E1

Beschrijving:

De inhoud van de geheugen locatie, geadresseerd door de stackpointer, wordt geladen in het lage orde deel van register IX. De stackpointer wordt met 1 verhoogd. De inhoud van het geheugen waar de stackpointer nu naar wijst, wordt in het hoge orde deel van IX geladen. De stackpointer wordt nog een keer met 1 verhoogd.

Datastroom:



Timing:

4 M cycli; 14 T states; 7 usec @ 2 MHz

Adresseringsmode:

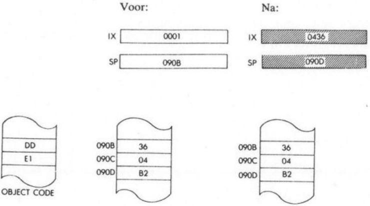
Indirect.

Flags:

S	Z		H		P/V	N	C

(geen invloed).

Voorbeeld: POP IX



POP IY

plaats de top van de stapel in IY

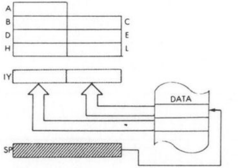
Funktie: $IY_{laag} \leftarrow (SP); IY_{hoog} \leftarrow (SP+1); SP \leftarrow SP + 2$

Formaat:

1	1	1	1	1	1	0	1	byte 1: FD
1	1	1	0	0	0	0	1	byte 2: E1

Beschrijving: De inhoud van de geheugen locatie, geadresseerd door de stackpointer, wordt geladen in het lage orde deel van register IY. De stackpointer wordt met 1 verhoogd. De inhoud van het geheugen waar de stackpointer nu naar wijst, wordt in het hoge orde deel van IY geladen. De stackpointer wordt nog een keer met 1 verhoogd.

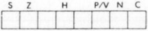
Datastroom:



Timing: 4 M cycli; 14 T states; 2 usec @ 2 MHz

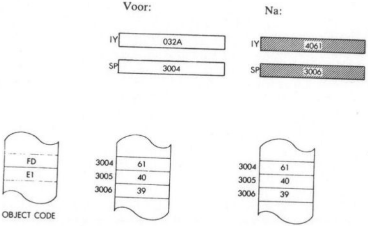
Adresseringsmode: Indirect.

Flags:



(geen invloed).

Voorbeeld: POP IY



PUSH qq

plaats register paar qq op de stapel

Funktie:

$(SP - 1) \leftarrow qq_{hoog}$ $(SP - 2) \leftarrow qq_{laag}$;
$SP \leftarrow SP - 2$

Formaat:

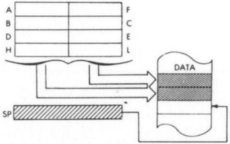
1	1	q	q	0	1	0	1

Beschrijving:

De stackpointer wordt met 1 verlaagd, en de inhoud van het hoge orde deel van het gespecificeerde register paar wordt geladen op het adres waar de stackpointer naar wijst. Opnieuw wordt de stackpointer met 1 verlaagd. De inhoud van het lage orde deel van het gespecificeerde register paar wordt geladen op het adres, waar de stackpointer nu naar wijst. QQ kan zijn:

BC - 00	HL - 10
DE - 01	AF - 11

Datastroom:



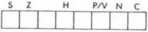
Timing: 3 M cycli; 11 T states; 6.5 usec @ 2 MHz

Adresseringsmode: Indirect.

Byte codes:

qq:	BC	DE	HL	AF
	C5	D5	E5	F5

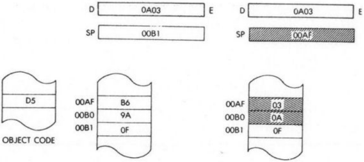
Flags:



(geen invloed).

Voorbeeld: PUSH DE

Voor: Na:



PUSH IX

plaats IX op de stapel

Funktie:

$(SP - 1) \leftarrow IX_{hoog}$; $(SP - 2) \leftarrow IX_{laag}$;
$SP \leftarrow SP - 2$

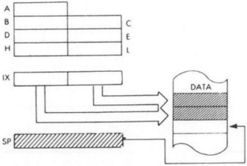
Formaat:

1	1	0	1	1	1	0	1	byte 1: DD
1	1	1	0	0	1	0	1	byte 2: E5

Beschrijving:

De stackpointer wordt met 1 verlaagd, en de inhoud van het hoge orde deel van het index register wordt geladen op het geheugen adres waar de stackpointer naar wijst. Opnieuw wordt de stackpointer verlaagd met 1, en het lage orde deel van het index register wordt geladen met de inhoud van het adres waar de stackpointer nu naar wijst.

Datastroom:



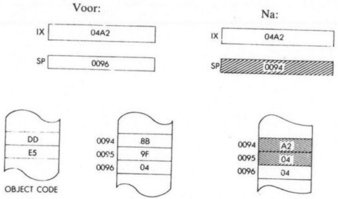
Timing: 4 M cycli; 15 T states; 7.5 usec @ 2 MHz

Adresseringsmode: Indirect.

Flags:

S	Z		H		P/V	N	C	
								(geen invloed).

Voorbeeld: PUSH IX



PUSH IY — plaats IY op de stapel

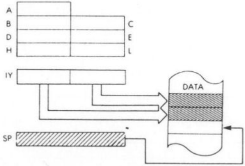
Funktie: $(SP - 1) \leftarrow IY_{hoog}$; $(SP - 2) \leftarrow IY_{laag}$;
$SP \leftarrow SP - 2$

Formaat:

1	1	1	1	1	1	0	1	byte 1: FD
1	1	1	0	0	1	0	1	byte 2: E5

Beschrijving: De stackpointer wordt met 1 verlaagd, en de inhoud van het hoge orde deel van het index register wordt geladen op het geheugen adres waar de stackpointer naar wijst. Opnieuw wordt de stackpointer verlaagd met 1, en het lage orde deel van het index register wordt geladen met de inhoud van het adres waar de stackpointer nu naar wijst.

Datastroom:



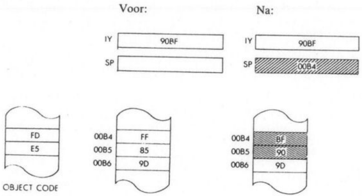
Timing: 3 M cycli; 15 T states; 7.5 usec @ 2 MHz

Adresseringsmode: Indirect.

Flags:

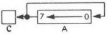
S	Z		H		P/V	N	C	
								(geen invloed).

Voorbeeld: PUSH IY



RLCA roteer de accumulator links

Funktie:



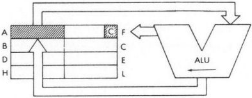
Formaat:

0	0	0	0	0	1	1	1

07

Beschrijving: De inhoud van de accumulator wordt 1 positie naar links geroteerd. De oorspronkelijke inhoud van bit 7 van de accumulator wordt zowel in de carry geplaatst, als in bit 0.

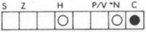
Datastroom:



Timing: 1 M cyclus; 4 T states; 2 usec @ 2 MHz

Adresseringsmode: Impliciet.

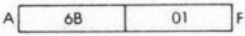
Flags:



C wordt geset door bit 7 van A.

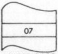
Voorbeeld: RLCA

Voor:



Na:





OBJECT CODE

RES b,s reset bit b van register s

Funktie: $s_b \leftarrow 0$

Formaat:

s:	7	6	5	4	3	2	1	0	
r	1	1	0	0	1	0	1	1	byte 1: CB
	1	0	←	b	→	←	r	→	byte 2
(HL)	1	1	0	0	1	0	1	1	byte 1: CB
	1	0	←	b	→	1	1	0	byte 2
(IX + d)	1	1	0	1	1	1	0	1	byte 1: DD
	1	1	0	0	1	0	1	1	byte 2: CB
	←			d				→	byte 3: verplaatsing
	1	0	←	b	→	1	1	0	byte 4
(IY + d)	1	1	1	1	1	1	0	1	byte 1: FD
	1	1	0	0	1	0	1	1	byte 2: CB
	←			d				→	byte 3: verplaatsing
	1	0	←	b	→	1	1	0	byte 4

b kan zijn:

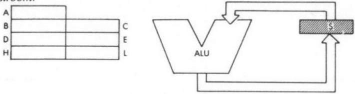
0 – 000	4 – 100
1 – 001	5 – 101
2 – 010	6 – 110
3 – 011	7 – 111

r kan zijn:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Beschrijving: Het door b gespecificeerde bit van de door s gespecificeerde register wordt gereset (0 gemaakt). S wordt gedefinieerd als bij de BIT instructie.

Datastroom:



Timing:

s:	M cycli:	T states:	usec @ 2 MHz:
r	2	8	4
(HL)	4	15	7.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Adresseringsmode: r: impliciet; (HL): indirect; (IX + d), (IY + d): geïndexeerd.

Byte codes: RES b, r

CB— b: \ r:	A	B	C	D	E	H	L
0	87	80	81	82	83	84	85
1	8F	88	89	8A	8B	8C	8D
2	97	90	91	92	93	94	95
3	9F	98	99	9A	9B	9C	9D
4	A7	A0	A1	A2	A0	A4	A5
5	AF	A8	A9	AA	AB	AC	AD
6	B7	B0	B1	B2	B3	B4	B5
7	BF	B8	B9	BA	BB	BC	BD

RES b, (HL)

b:	0	1	2	3	4	5	6	7
CB—	86	8E	96	9E	A6	AE	B6	BE

RES b, (IX + d)

RES b, (IY + d)

DD— / FD— CB —

b:	0	1	2	3	4	5	6	7
	86	8E	96	9E	A6	AE	B6	BE

Flags:

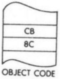
S	Z		H		P/V	N	C

(geen invloed).

Voorbeeld: RES 1, H

Voor: H 42

Na: H 40



RET terugkeer van subroutine

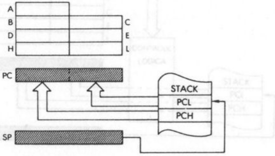
Funktie: $PC_{laag} \leftarrow (SP); PC_{hoog} \leftarrow (SP+1); SP \leftarrow SP+2$

Formaat:

1	1	0	0	1	0	0	1	C9

Beschrijving: De programma teller wordt van de stapel gehaald, zoals dat beschreven is bij de POP instructie. De volgende instructie wordt van het nieuwe adres in PC gehaald.

Datastroom:



Timing: 3 M cycli; 10 T states; 5 usec @ 2 MHz

Adresseringsmode: Indirect.

Flags:

S	Z		H		P/V	N	C	
								(geen invloed).

Byte codes:

CC: NZ	Z	NC	C	PO	PE	P	M
C0	C8	D0	D8	E0	E8	F0	F8

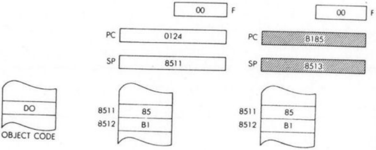
Flags:

S	Z		H		P/V	N	C

(geen invloed).

Voorbeeld: RET NC

Voor: Na:



RETI terugkeer van interrupt

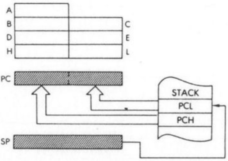
Funktie: $PC_{laag} \leftarrow (SP); PC_{hoog} \leftarrow (SP) + 1); SP \leftarrow SP + 2$

Formaat:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	0	0	1	1	0	1	byte 2: 4D

Beschrijving: De programma teller wordt van de stapel gehaald, zoals dat beschreven is bij de POP instructie. Deze instructie wordt door Zilog randapparatuur chips herkent als het einde van een interrupt routine, om het juiste verloop van geneste prioriteit interrupts te verzekeren. Opdat de processor volgende interrupts zou kunnen behandelen, moet RETI voorafgegaan worden door de EI instructie.

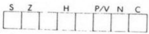
Datastroom:



Timing: 4 M cycli; 14 T states; 7 usec @ 2 MHz

Adresseringsmode: Indirect.

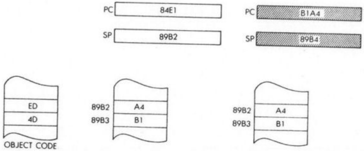
Flags:



(geen invloed).

Voorbeeld: RETI

Voor: Na:



RETN terugkeer van niet-maskeerbare interrupt

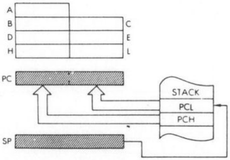
Funktie: $PC_{laag} \leftarrow (SP); PC_{hoog} \leftarrow (SP) + 1); SP \leftarrow SP + 2; IFF1 \leftarrow IFF2$

Formaat:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	0	0	0	1	0	1	byte 2: 45

Beschrijving: De programma teller wordt van de stapel gehaald, zoals dat beschreven is bij de POP instructie. Daarna wordt de inhoud van de IFF2 flip-flop in IFF1 geplaatst, om de oude status van de interrupt vlag te herstellen van voor de niet-maskeerbare interrupt.

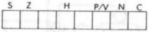
Datastroom:



Timing: 4 M cycli; 14 T states; 7 usec @ 2 MHz

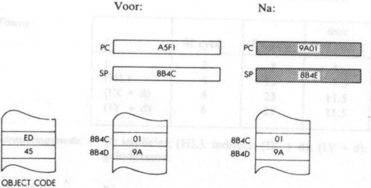
Adresseringsmode: Indirect.

Flags:



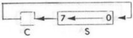
(geen invloed).

Voorbeeld: RETN



RL s — roteer operand s en de carry vlag links

Funktie:



Formaat:

s:									
r	1	1	0	0	1	0	1	1	byte 1: CB
	0	0	0	1	0	←	r	→	byte 2
(HL)	1	1	0	0	1	0	1	1	byte 1: CB
	0	0	0	1	0	1	1	0	byte 2: 16
(IX + d)	1	1	0	1	1	1	0	1	byte 1: DD
	1	1	0	0	1	0	1	1	byte 2: CB
	←			d				→	byte 3: Verplaatsing
	0	0	0	1	0	1	1	0	byte 4: 16
(IY + d)	1	1	1	1	1	1	0	1	byte 1: FD
	1	1	0	0	1	0	1	1	byte 2: CB
	←			d				→	byte 3: Verplaatsing
	0	0	0	1	0	1	1	0	byte 4: 16

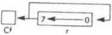
r kan zijn:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Beschrijving: De inhoud van het door de operand s gespecificeerde adres wordt 1 bit naar links geschoven. De inhoud van de carry komt in bit 0, en de carry vlag komt in bit 7. Het uiteindelijke resultaat wordt op het oorspronkelijke adres teruggeplaatst.

RLC r — roteer register r links

Funktie:



Formaat:

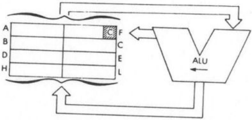
1	1	0	0	1	0	1	1	byte 1: CB

0	0	0	0	0	← r →	byte 2

Beschrijving: De inhoud van het gespecificeerde register wordt links geroteerd. De oorspronkelijke inhoud van bit 7 wordt naar de carry verplaatst en naar bit 0. R kan zijn:

A - 111
B - 000
C - 001
D - 010
E - 011
H - 100
L - 101

Datastroom:



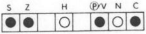
Timing: 2 M cycli; 8 T states; 4 usec @ 2 MHz

Adresseringsmode: Impliciet.

Byte codes: r:

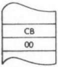
	A	B	C	D	E	H	L
CB-	07	00	01	02	03	04	05

Flags:



C wordt geset door bit 7 van r.

Voorbeeld: RLC B



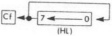
OBJECT CODE

Voor: Na:



RLC (HL) roteer geheugen plaats (HL) links

Funktie:

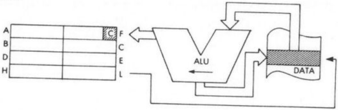


Formaat:

1	1	0	0	1	0	1	1	byte 1: CB
0	0	0	0	0	1	1	0	byte 2: 06

Beschrijving: De inhoud van de geheugen locatie, geadresseerd door de inhoud van register paar HL, wordt 1 bit positie links geroteerd. Het resultaat wordt op het zelfde adres terug geplaatst. De inhoud van bit 7 gaat naar het carry bit en bit 0.

Datastroom:



Timing: 4 M cycli; 15 T states; 7.5 usec @ 2 MHz

Adresseringsmode: Indirect.

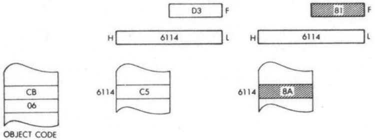
Flags:

S	Z		H		P/V	N	C
●	●		○		●	○	●

C wordt geset door bit 7 van de geheugen locatie.

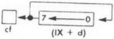
Voorbeeld: RLC (HL)

Voor: Na:



RLC (IX + d) roteer geheugen plaats (IX + d) links

Funktie:



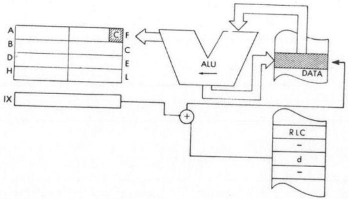
Formaat:

1	1	0	1	1	1	0	1	byte 1: DD
1	1	0	0	1	0	1	1	byte 2: CB
←			d				→	byte 3: Verplaatsing
0	0	0	0	0	1	1	0	byte 4: 06

Beschrijving:

De inhoud van het geheugen adres waar het index register plus de verplaatsing naar wijst wordt links geroteerd, en het resultaat wordt op het zelfde adres teruggeplaatst. De inhoud van bit 7 gaat naar het carry bit, en bit 0.

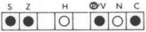
Datastroom:



Timing: 6 M cycli; 23 T states; 11.5 usec @ 2 MHz

Adresseringsmode: Geïndexeerd.

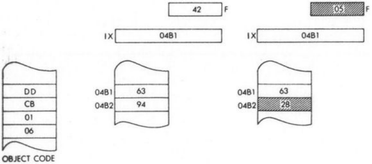
Flags:



C wordt geset door bit 7 van de geheugen locatie

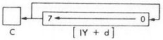
Voorbeeld: RLC (IX + 1)

Voor: Na:



RLC (IY + d) roteer geheugen plaats (IY + d) links

Funktie:



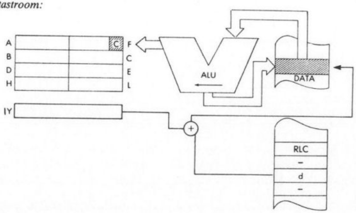
Formaat:

1	1	1	1	1	1	0	1	byte 1: FD
1	1	0	0	1	0	1	1	byte 2: CB
←			d				→	byte 3: Verplaatsing
0	0	0	0	0	1	1	0	byte 4: 06

Beschrijving:

De inhoud van het geheugen adres waar het index register plus de verplaatsing naar wijst wordt links geroteerd, en het resultaat wordt op het zelfde adres teruggeplaatst. De inhoud van bit 7 gaat naar het carry bit, en bit 0.

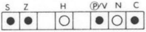
Datastroom:



Timing: 6 M cycli; 23 T states; 11.5 usec @ 2 MHz

Adresseringsmode: Geïndexeerd.

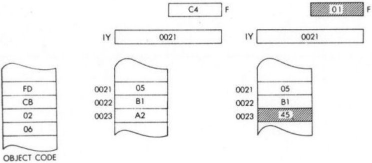
Flags:



C wordt geset door bit 7 van de geheugen locatie.

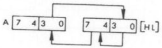
Voorbeeld: RLC (IY + 2)

Voor: Na:



RLD roteer links decimaal

Funktie:

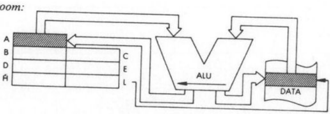


Formaat:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	1	0	1	1	1	1	byte 2: 6F

Beschrijving: De vier lage orde bits van het adres, waar HL naar wijst, worden verplaatst naar de vier hoge orde bits van het zelfde adres. De vier hoge orde bits worden verplaatst naar de vier lage orde bits van de accumulator. De vier lage orde bits van de accumulator worden verplaatst naar de vier lage orde bits van het al eerder gespecificeerde adres. Al deze verplaatsingen vinden tegelijk plaats.

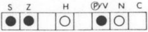
Datastroom:



Timing: 5 M cycli; 18 T states; 9 usec @ 2 MHz

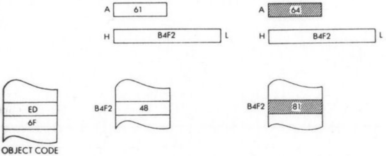
Adresseringsmode: Indirect.

Flags:



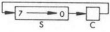
Voorbeeld: RLD

Voor: Na:



RR s roteer operand s en de carry vlag rechts

Funktie:



Formaat:

r	1	1	0	0	1	0	1	1	byte 1: CB
	0	0	0	1	1	←	r	→	byte 2
(HL)	1	1	0	0	1	0	1	1	byte 1: CB
	0	0	0	1	1	1	1	0	byte 2: 1E
(IX + d)	1	1	0	1	1	1	0	1	byte 1: DD
	1	1	0	0	1	0	1	1	byte 2: CB
	←			d				→	byte 3: Verplaatsing
	0	0	0	1	1	1	1	0	byte 4: 1E
(IY + d)	1	1	1	1	1	1	0	1	byte 1: FD
	1	1	0	0	1	0	1	1	byte 2: CB
	←			d				→	byte 3: Verplaatsing
	0	0	0	1	1	1	1	0	byte 4: 1E

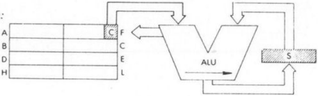
r kan zijn:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Beschrijving:

De inhoud van de door s gespecificeerde operand wordt naar rechts geschoven. De inhoud van de carry vlag komt in bit 7, en de inhoud van bit 0 komt in de carry vlag. Het resultaat wordt op het oorspronkelijke adres teruggeplaatst. S kan zijn, zoals gespecificeerd bij de beschrijving van de RLC instructie.

Datastroom:



Timing:

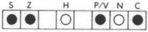
s:	*M cycli:*	*T states:*	*usec @ 2 MHz:*
r	2	8	4
(HL)	4	15	7.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Adresseringsmode: r: impliciet; (HL): indirect; (IX + d), (IY + d): geïndexeerd.

Byte codes: RR r:

r:	A	B	C	D	E	H	L
CB-	1F	18	19	1A	1B	1C	1D

Flags:

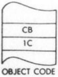


C wordt geset door bit 0 van de oorspronkelijke data.

Voorbeeld: RR H

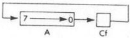
Voor: Na:





RRA roteer de accumulator en de carry vlag rechts

Funktie:



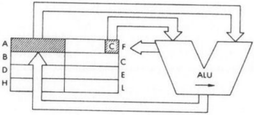
Formaat:

0	0	0	1	1	1	1	1

1F

Beschrijving: De inhoud van de accumulator wordt 1 positie naar rechts geschoven. De inhoud van de carry komt in bit 7, en de inhoud van bit 0 komt in de carry. (9 bits rotatie).

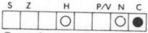
Datastroom:



Timing: 1 M cyclus; 4 T states; 2 usec @ MHz

Adresseringsmode: Impliciet.

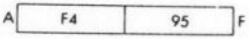
Flags:



C wordt geset door bit 0 van A.

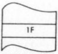
Voorbeeld: R R A

Voor:



Na:

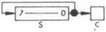




OBJECT CODE

RRC s roteer operand s rechts

Funktie:



Formaat: s: s kan zijn r, (HL), (IX + d), (IY + d)

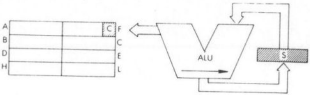
r	1	1	0	0	1	0	1	1	byte 1: CB
	0	0	0	0	1	←	r	→	byte 2
(HL)	1	1	0	0	1	0	1	1	byte 1: CB
	0	0	0	0	1	1	1	0	byte 2: 0E
(IX + d)	1	1	0	1	1	1	0	1	byte 1: DD
	1	1	0	0	1	0	1	1	byte 2: CB
	←			d				→	byte 3: Verplaatsing
	0	0	0	0	1	1	1	0	byte 4: 0E
(IY + d)	1	1	1	1	1	1	0	1	byte 1: FD
	1	1	0	0	1	0	1	1	byte 2: CB
	←			d				→	byte 3: Verplaatsing
	0	0	0	0	1	1	1	0	byte 4: 0E

r kan zijn:

A – 111	E – 011
B – 000	H – 100
C – 011	L – 101
D – 010	

Beschrijving: De inhoud van het geheugen adres, gespecificeerd door de operand, wordt rechts geroteerd, en het resultaat wordt op het zelfde adres teruggeplaatst. De inhoud van bit 0 gaat naar de carry, en naar bit 7. S wordt gedefinieerd bij de RLC instructie.

Datastroom:



Timing:

s:	*M cycli:*	*T states:*	*usec @ 2 MHz:*
r	2	8	4
(HL)	4	15	7.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Adresseringsmode: r: impliciet; (HL): indirect; (IX + d), (IY + d): geïndexeerd.

Byte codes: RRC r

r:	A	B	C	D	E	H	L
CB	0F	08	09	0A	0B	0C	0D

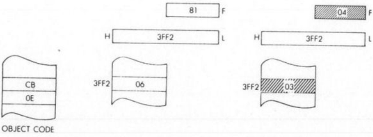
Flags:

S	Z		H		P/V	N	C
●	●		○		●	○	●

C wordt geset door bit 0 van de door s gespecificeerde data.

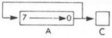
Voorbeeld: RRC (HL)

Voor: Na:



RRCA — roteer de accumulator rechts

Funktie:

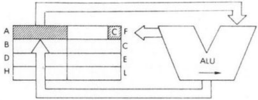


Formaat:

0	0	0	0	1	1	1	1	0F

Beschrijving: De inhoud van de accumulator wordt 1 positie rechts geroteerd. De inhoud van bit 0 gaat naar de carry, en naar bit 7.

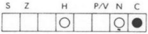
Datastroom:



Timing: 1 M cyclus; 4 T states; 2 usec @ 2 MHz

Adresseringsmode: Impliciet.

Flags:



C wordt geset door bit 0 van A.

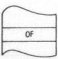
Voorbeeld: RRCA

Voor:

A	D4	51	F

Na:

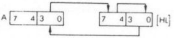
A	6A	40	F



OBJECT CODE

RRD roteer rechts decimaal

Funktie:



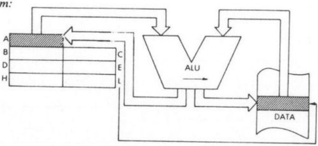
Formaat:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	1	0	0	1	1	1	byte 2: 67

Beschrijving:

De vier hoge orde bits van het adres waar (HL) naar wijst, worden verplaatst naar de vier lage orde bits op het zelfde adres. De vier lage orde bits gaan naar de vier lage orde bits van de accumulator. De vier lage orde bits van de accumulator gaan naar de vier hoge orde bits op het oorspronkelijke geheugen adres. Alle verplaatsingen vinden tegelijk plaats.

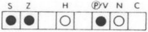
Datastroom:



Timing: 5 M cycli; 18 T states; 9 usec @ 2 MHz

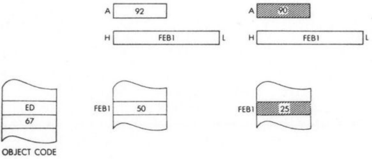
Adresseringsmode: Indirect.

Flags:



Voorbeeld: RRD

Voor: Na:



RST p

start opnieuw op p

Funktie:

$(SP-1) \leftarrow PC_{hoog}; (SP-2) \leftarrow PC_{laag}; SP \leftarrow SP - 2; PC_{hoog} \leftarrow 0; PC_{laag} \leftarrow p$

Formaat:

1	1	←	p	→	1	1	1

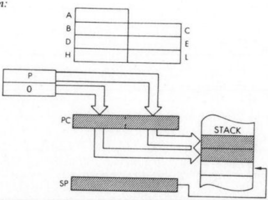
Beschrijving:

De inhoud van de programma teller wordt op de stapel geplaatst, zoals beschreven is bij de PUSH instructie. PC wordt geladen met de gespecificeerde waarde voor p. De volgende instructie komt van het nieuwe adres. P kan zijn:

00H - 000	20H - 100
08H - 001	28H - 101
10H - 010	30H - 110
18H - 011	38H - 111

Deze instructie springt naar een van de acht start adressen in het lage deel van het geheugen. De instructie is slechts 1 byte lang, en kan worden gebruikt als een snelle reactie op een interrupt.

Datastroom:



Timing: 3 M cycli; 11 T states; 5.5 usec @ 2 MHz

Adresseringsmode: Indirect.

Byte codes:

p:	00	08	10	18	20	28	30	38
	C7	CF	D7	DF	E7	EF	F7	FF

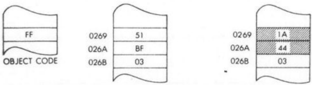
Flags:

S	Z		H		P/V	N	C

(geen invloed).

Voorbeeld: RST 38H

Voor:		Na:	
PC	441A	PC	0038
SP	026B	SP	0269



SBC A,s trek de gespecificeerde operand af van de accumulator, met borrow.

Funktie: $A \leftarrow A - s - C$

Formaat: *s:* kan zijn r, n, (HL), (IX + d), or (IY + d)

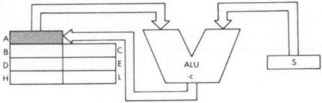
r	1 0 0 1 1 ←r→	
n	1 1 0 1 1 1 1 0	byte 1: DE
	←n→	byte 2: onmiddellijk data
(HL)	1 0 0 1 1 1 1 0	byte 1: 9E
(IX + d)	1 1 0 1 1 1 0 1	byte 1: DD
	1 0 0 1 1 1 1 0	9E
	←d→	byte 3: Verplaatsing
(IY + d)	1 1 1 1 1 1 0 1	byte 1: FD
	1 0 0 1 1 1 1 0	byte 2: 9E
	←d→	byte 3: Verplaatsing

r kan zijn:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Beschrijving: De gespecificeerde operand s, plus de inhoud van de carry, worden van de inhoud van de accumulator afgetrokken. Het resultaat komt in de accumulator. S wordt gedefinieerd bij de ADD instructie.

Datastroom:



Timing:

s:	M cycli:	T states:	usec @ 2 MHz:
r	1	4	2
n	2	7	3.5
(HL)	2	7	3.5
(IX + d)	5	19	9.5
(IY + d)	5	19	9.5

Adresseringsmode: r: impliciet; n: onmiddellijk; (HL): indirect; (IX + d), (IY + d): geïndexeerd.

Byte codes: SBC A, r

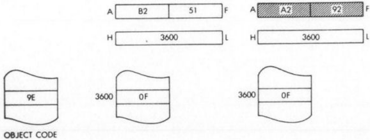
r:	A	B	C	D	E	H	L
	9F	98	99	9A	9B	9C	9D

Flags:

S	Z		H		P/V	N	C
●	●		●		●	1	●

Voorbeeld: SBC A, (HL)

Voor: Na:



SBC HL,ss

trek register paar ss af van HL, met borrow

Funktie: HL ← HL − ss − C

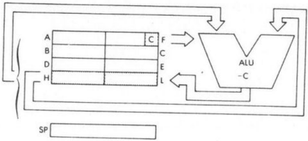
Formaat:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	s	s	0	0	1	0	byte 2

Beschrijving: De inhoud van het gespecificeerde register paar, plus de carry, worden van de inhoud van HL afgetrokken. Het resultaat komt in HL. SS kan zijn:

BC - 00	HL - 10
DE - 01	SP - 11

Datastroom:



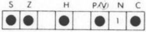
Timing: 4 M cycli; 15 T states; 7.5 usec @ 2 MHz

Adresseringsmode: Impliciet.

Byte codes:

ss:	BC	DE	HL	SP
ED--	42	52	62	72

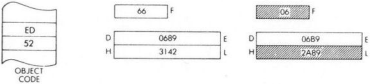
Flags:



Bit 12 set H bij een borrow
C wordt geset bij een borrow.

Voorbeeld: SBC HL, DE

Voor: Na:



SCF set carry vlag

Funktie: C ← 1

Formaat:

0	0	1	1	0	1	1	1	37

Beschrijving: De carry vlag wordt 1 gemaakt.

Timing: 1 M cyclus; 4 T states; 2 usec @ 2 MHz

Adresseringsmode: Impliciet.

Flags:

S	Z		H		P/V	N	C
			O			O	1

SET b,s set bit b van operand s

Funktie: $s_b \leftarrow 1$

Formaat: *s:*

	7	6	5	4	3	2	1	0	
r	1	1	0	0	1	0	1	1	byte 1: CB
	1	1	←	b	→	←	r	→	byte 2
(HL)	1	1	0	0	1	0	1	1	byte 1: CB
	1	1	←	b	→	1	1	0	byte 2
(IX + d)	1	1	0	1	1	1	0	1	byte 1: DD
	1	1	0	0	1	0	1	1	byte 2: CB
	←			d				→	byte 3: Verplaatsing
	1	1	←	b	→	1	1	0	byte 4
(IY + d)	1	1	1	1	1	1	0	1	byte 1: FD
	1	1	0	0	1	0	1	1	byte 2: CB
	←			d				→	byte 3: Verplaatsing
	1	1	←	b	→	1	1	0	byte 4

r kan zijn:

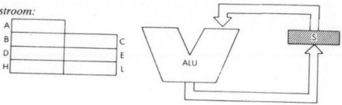
A – 111	E – 001
B – 000	H – 100
C – 001	L – 101
D – 010	

b kan zijn:

0 – 000	4 – 100
1 – 001	5 – 101
2 – 010	6 – 110
3 – 011	7 – 111

Beschrijving: Het gespecificeerde bit b van de inhoud van het door s gespecificeerde adres wordt geset. S wordt gedefinieerd bij de BIT instructie.

Datastroom:



Timing:

s:	*M cycli:*	*T states:*	*usec @ 2 MHz:*
r	2	8	4
(HL)	4	15	7.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Adresseringsmode: r: impliciet; (HL): indirect; (IX + d), (IY + d): geïndexeerd.

Byte codes: SET b, r

CB– b: \ r:	A	B	C	D	E	H	L
0	C7	C0	C1	C2	C3	C4	C5
1	CF	C3	C9	CA	CB	CC	CD
2	D7	D0	D1	D2	D3	D4	D5
3	DF	D8	D9	DA	DB	DC	DD
4	E7	E0	E1	E2	E3	E4	E5
5	EF	E8	E9	EA	EB	EC	ED
6	F7	F0	F1	F2	F3	F4	F5
7	FF	F8	F9	FA	FB	FC	FD

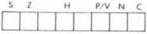
SET b, (HL)

SET b, (IX + d)

SET b, (IY + d)

b:	0	1	2	3	4	5	6	7
	C6	CE	Db	DE	E6	EE	F6	FE

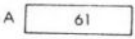
Flags:



(geen invloed).

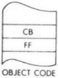
Voorbeeld: SET 7, A

Voor:



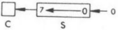
Na:





SLA s rekenkundige verschuiving naar links van operand s

Funktie:



Formaat: *s:*

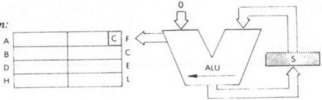
r	1	1	0	0	1	0	1	1	byte 1: CB
	0	0	1	0	0	← r →			byte 2
(HL)	1	1	0	0	1	0	1	1	byte 1: CB
	0	0	1	0	0	1	1	0	byte 2: 26
(IX + d)	1	1	0	1	1	1	0	1	byte 1: DD
	1	1	0	0	1	0	1	1	byte 2: CB
	← d →								byte 3: Verplaatsing
	0	0	1	0	0	1	1	0	byte 4: 26
(IY + d)	1	1	1	1	1	1	0	1	byte 1: FD
	1	1	0	0	1	0	1	1	byte 2: CB
	← d →								byte 3: Verplaatsing
	0	0	1	0	0	1	1	0	byte 4: 26

r kan zijn:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Beschrijving: De inhoud van de door s gespecificeerde operand wordt rekenkundig naar rechts geschoven. Bit 7 gaat naar de carry vlag, en bit 0 wordt 0. Het resultaat wordt op het zelfde adres geplaatst. S is gedefinieerd bij de RLC instructie.

Datastroom:



Timing:

s:	*M cycli:*	*T states:*	*usec @ 2 MHz:*
r	2	8	4
(HL)	4	15	7.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Adresseringsmode: r: impliciet; (HL): indirect; (IX + d), (IY + d): geïndexeerd.

Byte codes: SLA r

r:	A	B	C	D	E	H	L
CB	27	20	21	22	23	24	25

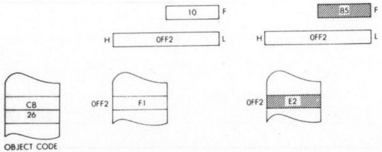
Flags:



C wordt geset door bit 7 van de verschoven data

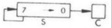
Voorbeeld: SLA (HL)

Voor: Na:



SRA s rekenkundige verschuiving naar rechts van operand s.

Funktie:



Formaat:

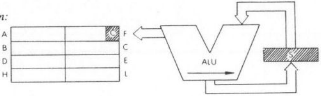
s:									
r	1	1	0	0	1	0	1	1	byte 1: CB
	0	0	1	0	1	←	r	→	byte 2
(HL)	1	1	0	0	1	0	1	1	byte 1: CB
	0	0	1	0	1	1	1	0	byte 2: 2E
(IX + d)	1	1	0	1	1	1	0	1	byte 1: DD
	1	1	0	0	1	0	1	1	byte 2: CB
	←			d				→	byte 3: Verplaatsing
	0	0	1	0	1	1	1	0	byte 4: 2E
(IY + d)	1	1	1	1	1	1	0	1	byte 1: FD
	1	1	0	0	1	0	1	1	byte 2: CB
	←			d				→	byte 3: Verplaatsing
	0	0	1	0	1	1	1	0	byte 4: 2E

r kan zijn:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Beschrijving: De inhoud van de door s gespecificeerde operand wordt rekenkundig naar rechts geschoven. De inhoud van bit 0 gaat naar de carry. De inhoud van bit 7 blijft gelijk. Het resultaat wordt op het zelfde adres geplaatst. S is gedefinieerd bij de RLC instructie.

Datastroom:



Timing:

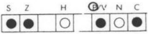
s:	M cycli:	T states:	usec @ 2 MHz:
r	2	8	4
(HL)	4	15	7.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Adresseringsmode: r: impliciet; (HL): indirect; (IX + d), (IY + d): geïndexeerd.

Byte codes: SRA r

r:	A	B	C	D	E	H	L
CB-	2F	28	29	2A	2B	2C	2D

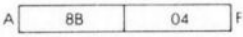
Flags:



C wordt geset door bit 0 van de verschoven data.

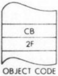
Voorbeeld: SRA A

Voor:



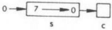
Na:





SRL s logische verschuiving naar rechts van operand s

Funktie:



Formaat:

s:									
r	1	1	0	0	1	0	1	1	byte 1: CB
	0	0	1	1	1	← r →			byte 2
(HL)	1	1	0	0	1	0	1	1	byte 1: CB
	0	0	1	1	1	1	1	0	byte 2: 3E
(IX + d)	1	1	0	1	1	1	0	1	byte 1: DD
	1	1	0	0	1	0	1	1	byte 2: CB
	← d →								byte 3: Verplaatsing
	0	0	1	1	1	1	1	0	byte 4: 3E
(IY + d)	1	1	1	1	1	1	0	1	byte 1: FD
	1	1	0	0	1	0	1	1	byte 2: CB
	← d →								byte 3: Verplaatsing
	0	0	1	1	1	1	1	0	byte 4: 3E

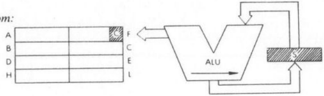
r kan zijn:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Beschrijving:

De inhoud van de door s gespecificeerde operand wordt logisch naar rechts geschoven. Een 0 wordt in bit 7 geschoven, en bit 0 komt in de carry. Het resultaat komt op het orginele adres te staan.

Datastroom:



Timing:

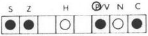
s:	M cycli:	T states:	usec @ 2 MHz:
r	2	8	4
(HL)	4	15	7.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Adresseringsmode: r: impliciet; (HL): indirect; (IX + d), (IY + d): geïndexeerd.

Byte codes: SRL r

r:	A	B	C	D	E	H	L
CB	3F	38	39	3A	3B	3C	3D

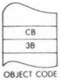
Flags:



C wordt geset door bit 0 van de verschoven data.

Voorbeeld: SRL E

Voor: Na:





SUB s trek operand s af van de accumulator

Funktie: A ← A − s

Formaat: *s:* kan zijn r, n, (HL), (IX + d) of (IY + d)

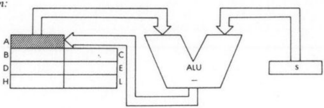
r	1	0	0	1	0	← r →			
n	1	1	0	1	0	1	1	0	byte 1: D6
	← n →								byte 2: onmiddellijk data
(HL)	1	0	0	1	0	1	1	0	96
(IX + d)	1	1	0	1	1	1	0	1	byte 1: DD
	1	0	0	1	0	1	1	0	byte 2: 96
	← d →								byte 3: Verplaatsing
(IY + d)	1	1	1	1	1	1	0	1	byte 1: FD
	1	0	0	1	0	1	1	0	byte 2: 96
	← d →								byte 3: Verplaatsing

r kan zijn:

A − 111	E − 011
B − 000	H − 100
C − 001	L − 101
D − 010	

Beschrijving: De gespecificerde operand s wordt van de accumulator afgetrokken, en het resultaat wordt in de accumulator geplaatst. S wordt gedefinieerd bij de ADD instructie.

Datastroom:



Timing:

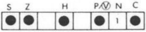
s:	M cycli:	T states:	usec @ 2 MHz:
r	1	4	2
n	2	7	3.5
(HL)	2	7	3.5
(IX + d)	5	19	9.5
(IX + d)	5	19	9.5

Adresseringsmode: r: impliciet; n: onmiddellijk; (HL): indirect; (IX + d), (IY + d): geïndexeerd.

Byte codes: SUB A, r

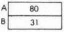
r:	A	B	C	D	E	H	L
	97	90	91	92	93	94	95

Flags:

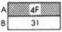


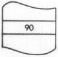
Voorbeeld: SUB A, B

Voor:



Na:





OBJECT CODE

XOR s

exclusieve of van accumulator en operand s

Funktie: $A \leftarrow A \veebar s$

Formaat: <u>s</u>: kan zijn r, n, (HL), (IX + d), of (IY + d)

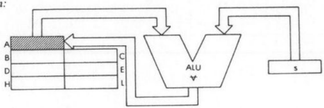
r	1	0	1	0	1	← r →			
n	1	1	1	0	1	1	1	0	byte 1: EE
	← n →								byte 2: onmiddellijk data
	1	0	1	0	1	1	1	0	AE
(IX + d)	1	1	0	1	1	1	0	1	byte 1: DD
	1	0	1	0	1	1	1	0	byte 2: AE
	← d →								byte 3: Verplaatsing
(IY + d)	1	1	1	1	1	1	0	1	
	1	0	1	0	1	1	1	0	
	← d →								

r kan zijn:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Beschrijving: De accumulator en de gespecificeerde operand s worden exclusief ge"of"t. Het resultaat wordt in de accumulator geplaatst. S wordt gedefinieerd bij de ADD instructies.

Datastroom:



Timing:

s:	*M cycli:*	*T states:*	*usec @ 2 MHz:*
r	1	4	2
n	2	7	3.5
(HL)	2	7	3.5
(IX + d)	5	19	9.5
(IY + d)	5	19	9.5

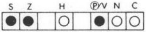
Adresseringsmode: r: impliciet; n: onmiddellijk; (HL): indirect; (IX + d), (IY + d): geïndexeerd.

Byte codes: XOR r

r:

A	B	C	D	E	H	L
AF	A8	A9	AA	AB	AC	AD

Flags:



Voorbeeld: XOR A, B1H

Voor: A 36

Na: A 87

5
ADRESSERINGS TECHNIEKEN

INLEIDING

In dit hoofdstuk komen de algemene theorieen betreffende het adresseren aan de orde, plus de verschillende technieken die ontwikkeld zijn voor het opslaan en het weer beschikbaar maken van data. In het tweede deel wordt een overzicht gegeven van de Z80 adresserings mogelijkheden, samen met hun voordelen en beperkingen. Tenslotte zullen, ten einde de lezer een beetje familiair te laten worden met het onderwerp, toepassingen worden gegeven.

De Z80 heeft behalve de programma teller meerdere 16-bits registers, die gebruikt kunnen worden, om een adres te specificeren. Daarom moet de Z80 gebruiker de verschillende adresserings mogelijkheden kennen, vooral die waarbij de index registers worden gebruikt. Complexe technieken kunnen we in het begin ter zijde laten liggen. Bij deze processor echter zijn alle adresserings technieken zeer bruikbaar tijdens het maken van programma's. Laten we de verschillende beschikbare mogelijkheden eens bekijken.

MOGELIJKE ADRESSERINGS TECHNIEKEN

Adresseren heeft, binnen een instructie, betrekking op de specificatie van de locatie van een operand, waarop de instructie een bewerking uitvoert. De belangrijkste adresserings methoden zullen nu worden behandeld. Zie ook figuur 5.1.

Impliciete adressering
(of "geimpliceerde" of "register" adressering)

Instructies die een bewerking uitvoeren op een register gebruiken normaal *impliciete adressering*. Dit is in figuur 5.1 geillustreerd. Deze instructies hebben deze naam gekregen, omdat ze geen adres bevatten. In plaats daarvan specificeert de opcode een of meer registers, meestal de accumulator, maar het kunnen ook andere registers zijn. Omdat het aantal interne registers beperkt is (het zijn er meestal 8), kan volstaan worden met drie bits voor de specificatie van het register. De totale instructie past normaal in 1 byte. Dat is een groot voordeel, want deze instructies zijn sneller uit te voeren dan 2- of 3-bytes instructies.

Een voorbeeld is:

LD A,B

welke de inhoud van B naar A verplaatst.

"Onmiddellijke" adressering

Immediate adressering staat eveneens geillustreerd in figuur 5.1. De 8-bits opcode wordt gevolgd door een 8- of 16-bits getal. Dit type instructie is, bijvoorbeeld, nodig om een 8-bits register te laden met een 8-bits getal. Omdat de processor ook 16-bits registers kent, kunnen ook 16-bits getallen geladen worden. Een voorbeeld van een immediate instructie is:

ADD A,0H

Het tweede woord van de instructie (0H) wordt bij A opgeteld.

Absolute adressering

Het opbergen van data in het geheugen en het er weer uit halen gebeurt normaal met absolute adressering. De opcode wordt gevolgd door een 16-bits adres. De instructie is dus drie bytes lang.

LD (1234H),A

is een voorbeeld van absolute adressering. De inhoud van de accumulator wordt op adres 1234H opgeslagen.

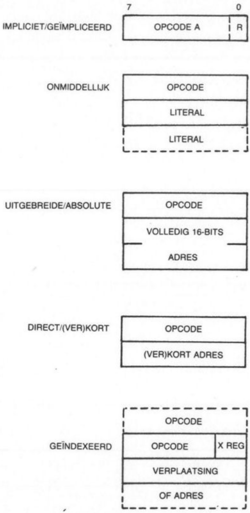


Fig. 5.1: Basis adresserings mogelijkheden

Het nadeel van dit type adressering is, dat de instructie drie bytes lang is. Om de efficientie van de processor groter te maken is een ander soort adressering uitgevonden: de directe adressering, die 1 byte gebruikt voor het adres.

Directe adressering
(of verkorte of relatieve adressering)

De opcode wordt gevolgd door een 8-bits adres. Zie figuur 5.1. Het voordeel van deze instructies is, dat ze slechts twee in plaats van drie bytes in beslag nemen. Het nadeel is het beperkte aantal mogelijke adressen (van 0 tot 255, of van −128 tot +127). Worden de adressen 0 tot 255 gebruikt, dan heet dat verkorte adressering, of "pagina nul" adressering. Absolute adressering wordt in dat opzicht vaak *uitgebreide adressering* genoemd. Het bereik van −128 tot +127 wordt gebruikt bij sprong instructies. Dan heet het relatieve adressering.

Relatieve adressering

Normale sprong instructies gebruiken 8 bits voor de opcode en 16 bits voor het sprongadres. Het nadeel hiervan is, net als bij het vorige voorbeeld, dat de instructie drie woorden groot is. Om een meer effectieve sprong mogelijk te maken, gebruikt de *relatieve adressering* slechts twee bytes. Het eerste byte geeft de specificatie van de soort sprong, vaak samen met een test. Het tweede byte is de verplaatsing. Omdat de verplaatsing positief en negatief kan zijn, liggen de grenzen vast: maximaal 127 adressen vooruit of 128 adressen terug (of eigenlijk +129 en −126, want PC is al verhoogd met 2). Loops, die meestal klein zijn, gebruiken vooral deze instructies, en worden er aanmerkelijk efficienter door. Een voorbeeld van een dergelijke instructie is (we hebben hem al gebruikt):

JR NC.

De twee voordelen van relatieve adressering zijn: een grotere snelheid, en het programma is reloceerbaar, d.w.z. het is onafhankelijk van absolute adressen.

Geindexeerde adressering

Geindexeerde adressering is een techniek die veel gebruikt wordt,

als op opeenvolgende elementen van een tabel een bewerking moet worden uitgevoerd. Dat zullen we later in dit hoofdstuk zien aan de hand van enkele voorbeelden. Het principe van dit soort adressering is, dat de instructie een register en een adres specificeert. Dat register is een index register. Het uiteindelijke adres ontstaat door de inhoud van het register bij het adres op te tellen. Het adres zou het beginadres kunnen zijn van een tabel in het geheugen. Het index register wordt dan gebruikt om op een efficiente manier achtereenvolgens alle elementen van die tabel te adresseren (daar zijn natuurlijk ook increment of decrement instructies voor nodig.) Vaak is de grootte van het index register of het adres beperkt.

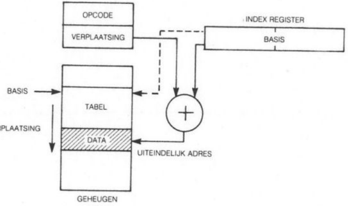


Fig. 5.2: Adressering (Pre-indexering)

Pre-indexering en post-indexering

Er zijn twee soorten indexering te onderscheiden. Bij pre-indexering vormt de som van de inhoud van het index register en het verplaatsings veld het echte adres. Zie figuur 5.2, met daarin een 8-bits verplaatsing en een 16-bits indexregister.

Post-indexering interpreteert de inhoud van het verplaatsings veld als het adres van de echte verplaatsing. Dit is in figuur 5.3 geillustreerd. Het is in feite een combinatie van indirecte adressering en pre-indexering. Maar we hebben indirecte adressering nog niet gedefinieerd. Laten we dat eerst eens gaan doen.

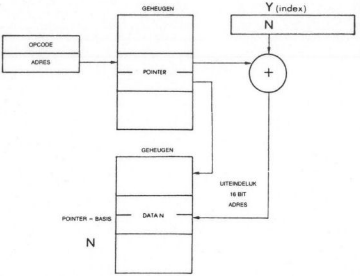


Fig. 5.3.: Indirecte geïndexeerde adressering (post-indexering)

Indirecte adressering

We hebben gezien, dat twee subroutines mogelijk grote hoeveelheden data in het geheugen wensen uit te wisselen. Of nog algemener: het zou mogelijk zijn dat meerdere programma's een zelfde blok informatie willen delen. Om zo'n programma zo algemeen mogelijk te houden, is het wenselijk dat dat blok niet vastzit op een plaats in het geheugen. Het blok moet dynamisch kunnen groeien of inkrimpen, en het zou op meerdere plaatsen in het geheugen kunnen zitten, afhankelijk van de grootte ervan. Absolute adressering is praktisch onmogelijk geworden met al deze eigenschappen.

De oplossing is: plaats het start adres op een vast adres. Dit lijkt op de situatie van een huis waarin meerdere personen wonen, en waarvan maar een sleutel bestaat. De sleutel wordt, volgens een gemaakte afspraak, onder de mat verstopt. Iedereen weet waar hij kijken moet om het huis in te kunnen. Indirecte adressering gebruikt een opcode plus een 16-bits adres. Dat adres wordt gebruikt om een 16-bits getal uit het geheugen te halen. Dat getal is het echte adres van de data die we wil-

len hebben. Zie figuur 5.4. De instructie bevat adres A1. Op dat adres staat een getal A2. A2 wordt als adres opgevat. Dat adres bevat de gewenste data.

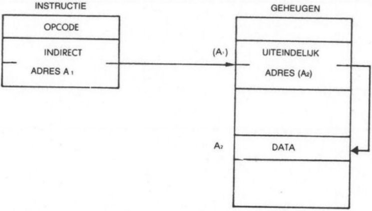


Fig. 5.4: Indirecte adressering

Indirecte adressering is bijzonder bruikbaar als pointers worden gebruikt. Verschillende delen van het programma kunnen dan verwijzen naar deze pointers. Op deze manier kunnen blokken data gemakkelijk en elegant door verschillen routines worden geadresseerd. Het uiteindelijke adres kan ook in een register zitten. De instructie wijst in dat geval naar dat register. Deze manier van adresseren heet "register indirect".

Combinaties

De bovenstaande adresserings mogelijkheden kunnen worden gecombineerd. Zo zou adres A2 in het vorige voorbeeld opgevat kunnen worden als een indirect adres, enz.

Geindexeerde adressering kan gecombineerd worden met indirecte adressering. Daardoor kan efficient toegang tot woord n van een blok data worden verkregen, aangenomen dat de pointer naar het startadres bekend is. Zie figuur 5.2.

We kennen nu de meeste soorten adressering die in een systeem aanwezig kunnen zijn. De meeste microprocessors bezitten niet alle moge-

lijkheden, en hebben slechts een deelverzameling van alle adresserings mogelijkheden. De Z80 bezit ook een deelverzameling ervan. Een goede. We gaan ze nu bekijken.

Z80 ADRESSERINGS MOGELIJKHEDEN

Geimpliceerde adressering (Z80)

Deze adressering wordt essentieel gebruikt door 1-bytes instructies die bewerkingen uitvoeren op interne registers. Als impliciete instructies uitsluitend met een intern register werken, zijn ze slechts 1 cyclus lang.

Voorbeelden van instructies met geimpliceerde (of register) adressering zijn: LD r,r'; ADD A,r; ADC A,r; SUB s; SBC A,s; AND s; OR s; XOR s; CP s en INC r.

Zilog maakt verder onderscheid tussen register adressering en geimpliceerde adressering. Geimpliceerde adressering is dan beperkt, volgens hun definitie, tot instructies zonder specifiek veld om een intern register aan te wijzen. Daarmee is nog een adresserings mogelijkheid meer geintroduceerd. Daarom is het aantal adresserings mogelijkheden alleen niet voldoende om de capaciteiten van een microprocessor aan te duiden.

Onmiddellijke adressering (Z80)

Omdat de Z80 twee soorten registers heeft, 8-bits en 16-bits groot, heeft deze processor ook twee soorten immediate adressering. Instructies zijn twee of drie bytes lang. Het tweede (en derde) byte bevat de constante welke in een register geladen moet worden, of waarmee de bewerking wordt uitgevoerd. Uitzonderingen zijn LD IX en LD IY, die een 16-bits opcode hebben.

Voorbeelden zijn:

```
LD   r,n     (twee bytes)
LD   dd,nn   (drie bytes)
ADD  A,n     (twee bytes)
```

Als de constante twee bytes lang is wordt de adressering "immediate uitgebreid" genoemd bij de Z80.

Absolute of"uitgebreide" adressering (Z80)

Per definitie heeft absolute adressering drie bytes nodig. Het eerste byte is de opcode en de volgende bytes het 16-bits adres.

In tegenstelling tot de verkorte adressering wordt dit ook wel de uitgebreide adressering genoemd.

Voorbeelden van deze adressering zijn:

```
LD HL,(nn)
JP  nn.
```

nn is een 16-bits adres en (nn) de inhoud van adres nn.

Gemodificeerde pagina nul adressering (Z80)

Pagina-nul adressering is behalve d.m.v. de call instructie niet beschikbaar op de Z80. De instructie gebruikt een gewijzigde versie, en heet dan: gemodificeerde pagina-nul adressering.

De CALL instructie heeft een drie-bits groot veld (bits 3, 4 en 5), dat gebruikt wordt om naar een van acht adressen op pagina 0 van het geheugen te wijzen. Het effectieve adres is b5b4b3000 en wordt in PC geladen. De instructie is snel, want hij is maar een byte groot. De instructie werd hoofdzakelijk gebruikt als reactie op meervoudige interrupts. Het nadeel daarvan is de beperkte hoeveelheid beschikbare ruimte voor de interrupt routine. Een sprong in die routine doet het voordeel weer teniet. De ruimte is beperkt, omdat de afstand tussen de routine adressen slechts 16 bytes is.

De instructie wordt voor dat doel steeds minder gebruikt, omdat tegenwoordig speciale "prioriteit interrupt controller" chips (PIC's) op de markt te verkrijgen zijn. Deze zorgen voor een juiste en snelle afhandeling van interrupts.

De instructie wordt nu meestal gebruikt voor "restarts".

Relatieve adressering (Z80)

Per definitie heeft relatieve adressering twee bytes nodig. Het eerste byte bevat de opcode, het tweede de verplaatsing met het teken.

Om de instructie te onderscheiden van de absolute sprong wordt de afkorting "JR" gebruikt.

De tijd nodig om de instructie uit te voeren kan verschillen. We moe-

ten daar met berekeningen voorzichtig mee zijn. Als niet aan de test wordt voldaan, dan heeft de instructie slechts zeven "T-states" nodig. De programma teller bevat dan al het adres van de volgende instructie. Wordt echter wel aan de test voldaan, m.a.w. als er gesprongen wordt, dan heeft de instructie 12 "T-states" nodig. Het nieuwe adres moet eerst worden berekend, waarna het in de programma teller geladen moet worden.

Bij het berekenen van de tijd die nodig is om een deel van een programma uit te voeren, moet met bovenstaande wel degelijk rekening worden gehouden.

Daarom wordt een loop sneller, bij gebruik van JR, als een conditie wordt getest waaraan meestal *niet* wordt voldaan.

Wordt JR buiten een loop gebruikt, dan wordt voor de berekening van de snelheid een gemiddelde waarde aangehouden.

Dit probleem doet zich niet voor bij de onvoorwaardelijke sprong JR e. De instructie test geen enkele voorwaarde en duurt altijd 12 "T-states".

Geindexeerde adressering (Z80)

Deze adresseer methode bestond niet bij de 8080, en werd er bij de Z80 aan toegevoegd (net als twee index registers). Daardoor moest de opcode een byte groter worden. Deze is dan 16 bits groot. (Een ander voorbeeld van een 16-bits opcode is LDIR) De structuur van een geindexeerde instructie staat in figuur 5.5.

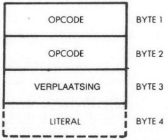


Fig. 5.5: De geindexeerde instructie heeft een 16-bits opcode

Instructies waarbij geindexeerde adressering is toegestaan zijn:

LD, ADD, INC, RLC, BIT en SET.

Deze adresserings methode wordt zeer veel gebruikt bij bewerkingen van blokken data, tabellen en lijsten.

Indirecte adressering (Z80)

De Z80 heeft een beperkte mogelijkheid tot indirect adresseren: de "register indirecte adressering". Ieder van de register paren BC, DE en HL kan als geheugen adres worden gebruikt.

Als ze naar 16-bits data wijzen, wijzen ze naar het minst significante deel ervan. Het meest significante deel zit in het volgende adres.

Combinaties

Combinaties bestaan eigenlijk niet, behalve dat instructies met twee operanden verschillende adresseringen mogen toepassen voor beide operanden.

Een *load* of een rekenkundige instructie kan de ene operand dus ophalen met immediate adressering, en de andere met geindexeerde adressering.

Op gelijke wijze kan het bit adresserings mechanisme werken met een van de drie hierboven genoemde adresserings mogelijkheden.

De adresserings mogelijkheden die bij iedere instructie gebruikt kunnen worden, staan in het voorgaande hoofdstuk.

Bit adressering

Bit adressering wordt meestal niet als een adresserings mogelijkheid opgevat. Maar hoe het ook wordt opgevat, bit adressering is een waardevolle faciliteit. Aangezien het door Zilog als een "adresserings mode" wordt gedefinieerd, zullen we het hier als zodanig opvatten. De 8080 was niet voorzien van deze adressering mode.

M.b.v. bit adressering kunnen gespecificeerde bits worden geadresseerd. De Z80 kan zodoende bepaalde bits setten, resetten en testen op een geheugenadres of een register. Het betreffende byte kan door "register", "register-indirect" of "geindexeerde" adressering worden geadresseerd. M.b.v. drie bits in de opcode wordt het juiste bit bepaald.

HET GEBRUIK VAN DE Z80 ADRESSERINGS MOGELIJKHEDEN

Lange en verkorte adressering

We hebben al gebruik gemaakt van de relatieve sprong instructies in de verschillende programma's. Deze toepassingen spreken voor zich zelf. Een interessante vraag is: wat moeten we doen, als het toegestane sprong bereik voor ons niet voldoende is? Een eenvoudige oplossing is de zogenaamde *lange sprong*. Dat is een sprong naar een adres, waar een absolute of lange sprong specificatie is geplaatst:

```
        JR    NC, $ + 3   SPRING ALS C = 0 NAAR
                          HET HUIDIGE ADRES + 3
        JP    FAR         SPRING ANDERS NAAR FAR
                          (VOLGENDE INSTRUCTIE)
```

Als de carry 1 is, wordt dus naar FAR gesprongen. Dit lost ons probleem op. De meer complexe adresseer methoden kunnen nu aan bod komen: geindexeerd en indirect.

Indexering gebruikt bij sequentieel data blok

Indexering wordt in de eerste plaats gebruikt om opeenvolgende adressen in een tabel te adresseren. De lengte mag niet groter zijn dan 256, opdat de verplaatsing in een 8-bits register zou passen.

We hebben al geleerd hoe we moeten testen op een karakter. In een tabel bestaande uit 100 elementen, moet een "*" gezocht worden. Het programma staat hierna. Zie ook het stroomdiagram in figuur 5.6.

```
SEARCH    LD    IX, BASE
          LD    A, '*'
          LD    B, COUNT
TEST      CP    (IX)
          JR    Z, FOUND
          INC   IX
          DEC   B
          JR    NZ, TEST
NOTFND    ...
```

In de paragraaf over blok verplaatsingen zullen we nog een verbeterd programma tegenkomen.

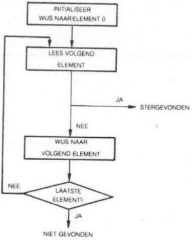


Fig. 5.6: Karakter zoek programma. Stroomdiagram

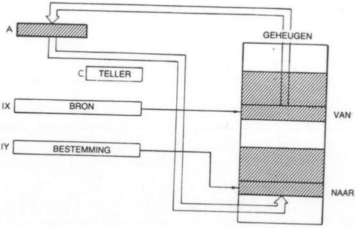


Fig. 5.7: Blok verplaatsing. Initialisering van het register

Een blok verplaats routine voor minder dan 256 elementen

"COUNT" is het aantal elementen van het te verplaatsen blok. We nemen aan dat dat aantal kleiner dan 256 is. FROM is het basis adres van het blok. TO is het basis adres in het geheugen, waar het blok naar verplaatst moet worden. Het algoritme is erg simpel: er wordt een woord tegelijk verplaatst. Teller C houdt bij, welk woord verplaatst wordt. Hier volgt het programma:

```
BLKMOV   LD   IX, FROM
         LD   IY, TO
         LD   C, COUNT
NEXT     LD   A, (IX)        HAAL WOORD
         LD   (IY), A
         INC  IX
         INC  IY
         DEC  C
         JR   NZ, NEXT
```

Laten we het programma eens nader bekijken:

```
BLKMOV   LD   IX,FROM
         LD   IY,TO
         LD   C, COUNT
```

Deze drie registers initialiseren de registers IX, IY, en C, zoals in figuur 5.7 is te zien.
Index register IX is de pointer naar de bron, en wordt regelmatig met 1 verhoogd. IY wijst naar de bestemming en wordt ook regelmatig met 1 verhoogd. Register C wordt geladen met het maximale aantal elementen dat verplaatst moet worden. Omdat het een 8-bits register is, is het maximale aantal 256. Van dit register wordt regelmatig 1 afgetrokken. Als C 0 wordt, dan zijn alle elementen verplaatst.
De volgende twee instructies:

```
NEXT     LD   A, (IX)
         LD   (IY), A
```

laden de inhoud van het geheugenadres waar IX naar wijst in A. Daarna wordt de inhoud van A verplaatst naar het adres waar IY naar wijst. Met andere woorden, er is een element van het ene blok naar het andere verplaatst. De twee index registers worden daarna met 1 verhoogd:

```
INC  IX
INC  IY
```

En de teller wordt met 1 verlaagd:

```
DEC  C
```

Tenslotte wordt, als C niet 0 is, terug gesprongen naar NEXT:

```
JR   NZ, NEXT
```

In dit voorbeeld wordt een toepassing van de index registers gegeven. Laten we dat programma eens vergelijken met een soortgelijk programma voor een andere microprocessor: de MOS technology 6502, welke ook indexering kent, maar met andere afspraken:

```
        LDX   #NUMBER
NEXT    LDA   FROM, X
        STA   TO, X
        DEX
        BNE   NEXT
```

Zonder op de details van het programma in te gaan, zal de lezer onmiddelijk zien, dat dit programma korter is. Dat komt, omdat het index register X als een variabele verplaatsing wordt gebruikt, terwijl BASE en DEST de vaste basis adressen van de blokken zijn.

Dit voorbeeld moet duidelijk maken, dat hoewel indexering in theorie een krachtige faciliteit is, het niet noodzakelijk tot een efficiente code leidt. Dat hangt af van de beperkingen van de verschillende microprocessors. Een echte algemene indexering heeft de mogelijkheid van een 16-bits verplaatsing of adres veld nodig, plus een 16-bits index register.

Dat probleem is bij de Z80 opgelost, door de aanwezigheid van enkele gespecialiseerde instructies. We zullen nu een algemene blok verplaatsing beschrijven, die slechts vier instructies lang is. Maar eerst nog enkele opgaven.

Oefening 5.1: Schrijf een programma om een blok te verplaatsen in de stijl van bovenstaand 6502 programma. D.w.z. het index register bevat de verplaatsing. Beide blokken bevinden zich op pagina 0, die loopt van adres 0 tot adres 256. Het aantal elementen is zo klein, dat de blokken elkaar niet overlappen.

Oefening 5.2: Neem nu aan, dat de blokken op een willekeurige plaats in het geheugen staan. De blokken bevinden zich beide op de zelfde pagina. Herschrijf bovenstaand programma.

Algemene blok verplaats routine (meer dan 256 elementen)

Figuur 5.8 geeft de toewijzing van de registers en een plattegrond van het geheugen. Het programma is:

```
LD     BC, COUNT     AANTAL BYTES
LD     DE, TO        ADRES BESTEMMING
LD     HL, FROM      START ADRES
LDIR                 VERPLAATS ALLE BYTES
```

Gebruikte geheugen ruimte: 11 bytes
Benodigde tijd: 21 cycli/byte

De eerste instructie is:

```
LD     BC, COUNT
```

In het register paar BC wordt het aantal (16 bit) van de te verplaatsen elementen geplaatst. De volgende twee instructies initialiseren de register paren DE en HL:

```
LD     DE, TO
LD     HL, FROM
```

Tenslotte:

```
LDIR
```

welke de hele verplaatsing uitvoert.

LDIR is een automatische blok verplaats instructie. LDIR doet het volgende: De inhoud van het adres waar HL naar wijst, gaat naar het adres waar DE naar wijst: (DE) = (HL). Vervolgens: DE = DE + 1 en HL = HL + 1. Daarna: BC = BC − 1. Als BC 0 wordt, wordt de instructie beeindigd. Anders wordt de instructie herhaald.

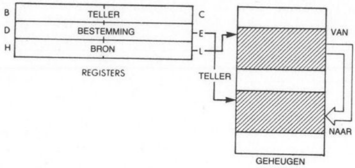


Fig. 5.8: Een blok verplaatsing. Plattegrond van het geheugen.

De waarde en de kracht van de LDIR instructie moet nu wel duidelijk zijn. Op soortgelijke wijze kan ons zoek programma door een automatische instructie worden verbeterd. Daarvoor gebruiken we CPIR, een speciale Z80 instructie. Het programma volgt nu:

```
        LD A,'*
        LD BC, COUNT
        LD HL, STRING
STAR    CPIR
        JR Z, STAR
NOSTAR  ---
```

De eerste instructie laadt de accumulator met de code voor de ster. Daarna wordt het register paar BC geinitialiseerd met het aantal te onderzoeken woorden in het blok:

```
LD BC, COUNT
```

In HL wordt het start adres van het blok geladen. De automatische instructie wordt daarna uitgevoerd:

```
LD HL, STRING
CPIR
```

De CPIR instructie is een automatische vergelijk instructie. De inhoud van het adres in HL wordt vergeleken met de inhoud van de accumula-

tor. Is de vergelijking succesvol, dan wordt de Z vlag 1 gemaakt. HL wordt daarna 1 groter, en BC 1 kleiner. De instructie wordt herhaald, tot BC 0 is, of tot de vergelijking succesvol is. Als de instructie CPIR voltooid is, moet de Z vlag dus getest worden om erachter te komen of de ster in het blok voorkomt. Als de ster niet in het blok voorkomt, kan de CPIR instructie in het extreme geval 64K geheugen hebben onderzocht. Het testen van de vlag gaat als volgt:

```
JR  Z, STAR
```

Oefening 5.3: Herschrijf het programma, zodat het blok achterwaarts wordt doorlopen tijdens het zoeken. (Hint: gebruik CPDR)

We gaan nu een programma maken, dat een combinatie is van de twee vorige programma's. Een blok data moet van FROM naar TO worden verplaatst. De verplaatsing moet automatisch stoppen als het begrenzingskarakter, "*", wordt gevonden.

```
        LD    BC,COUNT
        LD    HL,FROM
        LD    DE,TO
        LD    A,'*             BEGRENZINGSKARAKTER
                               (DELIMITER)
TEST    CP    (FROM)           VERGELIJK ACCU MET
                               GEHEUGEN
        JR    Z, END           EINDIGEN ALS TEST SUCCESVOL
        LDI                    VERPLAATS KAR. EN MAAK
                               POINTERS EN TELLER UP TO
                               DATE
        JR    PE, TEST         BLIJF TESTEN. P GEEFT AAN
                               OF BC = 0
```

De gebruikelijke initialisering wordt door de eerste drie instructies uitgevoerd. De teller, bron en bestemming registers krijgen de juiste waarde:

```
LD  BC, COUNT
LD  HL, FROM
LD  DE, TO
```

De ster wordt in de accumulator geladen:

```
        LD  A, '*
```

De volgende instructie vergelijkt het geheugen met de inhoud van de accumulator:

```
TEST    CP (FROM)
```

De Z vlag geeft aan of de test slaagt of niet. Deze vlag wordt 1 als de test slaagt. De volgende instructie test de vlag:

```
        JR  Z, END
```

De volgende instructie is een automatische verplaats instructie:

```
        LDI
```

Deze instructie verplaatst het karakter en werkt de pointers en de teller bij. LDI verplaatst de inhoud van het geheugen op adres (HL) naar het adres (DE), (DE) = (HL). Daarna wordt bij HL en DE 1 opgeteld:

DE = DE + 1
HL = HL + 1

Tenslotte wordt van BC 1 afgetrokken:

BC = BC − 1

Het eigenaardige van de instructie is, dat de P/V vlag geset wordt als BC 0 wordt, en gereset als BC niet 0 wordt. De volgende instructie test deze vlag, en beslist of het programma beeindigd moet worden:

```
        JR  PE, TEST
```

Het optellen van twee blokken

Het volgende programma telt element voor element twee blokken bij elkaar op, te beginnen bij de adressen BLK1 en BLK2. De twee blokken hebben het zelfde aantal elementen, COUNT.

```
BLKADD  LD    IX, BLK1
        LD    IY, BLK2
        LD    B, COUNT
        XOR   A
```

```
LOOP    LD    A, (IX + 0)
        ADC   A, (IY + 0)
        LD    (IX), A
        DEC   IX
        DEC   IY
        DEC   B
        JR    NZ, LOOP
```

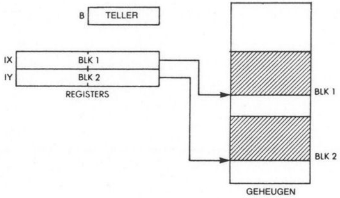


Fig. 5.9: Optelling van twee blokken: BLK1 = BLK1 + BLK2

Een plattegrond van het geheugen staat in figuur 5.9. Het programma is recht-toe-recht-aan. Het aantal elementen komt in B en IX en IY worden geinitialiseerd met de waarden BLK1 en BLK2:

```
BLKADD   LD  IX, BLK1
         LD  IY, BLK2
         LD  B, COUNT
```

Het carrybit wordt met het oog op de eerste optelling 0 gemaakt:

```
         XOR  A
```

Het eerste element wordt in de accumulator geladen:

```
LOOP     LD  A, (IX + 0)
```

Het corresponderende element van BLK2 wordt erbij opgeteld:

```
ADC  A, (IY +0)
```

en het resultaat komt in BLK1:

```
LD  (IX), A
```

Van de twee pointers wordt 1 afgetrokken:

```
DEC  IX
DEC  IY
```

evenals van de teller:

```
DEC  B
```

Zolang het tel register niet 0 is, wordt de loop uitgevoerd:

```
JR  NZ, LOOP
```

Oefening 5.4: Kun je bovenstaand programma gebruiken voor een 32-bits optelling?

Oefening 5.5: Kan het gebruikt worden voor een 64-bits optelling?

Oefening 5.6: Verander het programma zodanig, dat de resultaten in een apart blok komen, te beginnen bij adres BLK3.

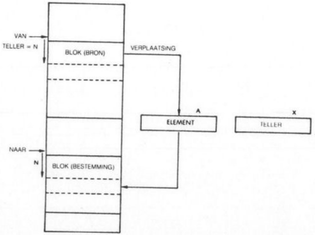


Fig. 5.10: Geheugen organisatie voor blok verplaatsing

Oefening 5.7: Verander het programma zodanig, dat er afgetrokken wordt i.p.v. opgeteld.

Oefening 5.8: Verander het programma zo, dat BLK1 en BLK2 wijzen naar de top van het blok i.p.v. naar de bodem ervan. Zie figuur 5.10.

SAMENVATTING

Er is in dit hoofdstuk een complete beschrijving van de adresserings mogelijkheden gegeven. De Z80 biedt de gebruiker vele adresserings technieken, die we geanaliseerd hebben. Tenslotte is d.m.v. programma's de waarde van de diverse mogelijkheden aangetoond. Om de Z80 efficient te programmeren is kennis van alle adresserings technieken een vereiste. Ze worden overal in de rest van het boek gebruikt.

Oefeningen

5.9: Schrijf een programma dat de eerste 10 bytes van een tabel op adres "BASE" optelt. Het resultaat heeft 16 bits. (Dit is een berekening van een "checksum")

5.10: Kun je het zelfde probleem oplossen zonder de geindexeerde adressering te gebruiken?

5.11: Draai de volgorde van de 10 bytes van de tabel om. Sla het resultaat op op adres "REVER".

5.12: Zoek in de zelfde tabel het grootste getal op. Laad dat getal in adres "LARGE".

5.13: Tel de corresponderende elementen van drie tabellen, die beginnen op de adressen BASE1, BASE2 en BASE3, bij elkaar op. De lengte van deze tabellen staat op adres "LENGTH" op pagina 0.

6

INPUT/OUTPUT TECHNIEKEN

INLEIDING

Tot nu toe hebben we geleerd, hoe we informatie tussen het geheugen en de verschillende registers van de microprocessor kunnen uitwisselen. We hebben ook geleerd hoe we de registers en de verschillende instructies moeten gebruiken om de data te manipuleren. In dit hoofdstuk komt de communicatie met de buitenwereld aan de orde, de input/output.

Input slaat op het naar binnen halen van data afkomstig van randapparatuur, zoals toetsenbord, en schijf. *Output* heeft betrekking op het zenden van data van de microprocessor of het geheugen naar de buitenwereld. dat kan zijn een printer, beeldscherm (CRT), schijf, relais enz.

Allereerst worden de I/O bewerkingen bekeken, die nodig zijn voor ieder randapparaat. Daarna zullen we zien hoe we meerdere randapparaten tegelijk kunnen bedienen. Vooral "polling" en "interrupts" zullen we onder de loep nemen.

INPUT/OUTPUT

In deze paragraaf wordt ons geleerd, hoe we eenvoudige signalen, zoals pulsen, kunnen ontvangen en opwekken. Daarna komen technieken aan de orde, waarmee pulsen van een bepaalde lengte kunnen worden opgewekt en getest. Is dat achter de rug, dan kunnen we ons bezig houden met meer ingewikkelde soorten I/O, zoals seriele en parallelle data overdracht met hoge snelheid.

De Z80 input/output instructies

De Z80 heeft een speciale set input en output instructies. De meeste processors hebben deze instructies niet, en gebruiken de normale instructies voor het doen van I/O. Evenals de 8080 heeft de Z80 basis I/O instructies. Bovendien heeft de Z80 enkele aanvullende instructies. Ze zullen gedetailleerd worden besproken.

De basis input en output instructies zijn respectivelijk: IN A,(n) en OUT A,(n). Deze instructies vormen de erfenis van de 8080. Ze lezen en schrijven respectivelijk een byte tussen de geselecteerde poort en de accumulator. Het adresseren van een poort gaat als volgt: het adres van de poort komt op de adreslijnen A0 tot en met A7 te staan. De inhoud van de accumulator staat op de adreslijnen A8 tot en met A15. Als deze laatste adreslijnen worden gedecodeerd door een I/O apparaat, is het soms nodig allereerst de inhoud van de accumulator nul te maken. In de volgende eenvoudige voorbeelden wordt aangenomen, dat deze adreslijnen niet met een I/O poort zijn verbonden, zodat we daar geen rekening mee hoeven te houden.

Bij de speciale instructie IN r,(C) wordt de inhoud van register C gebruikt als I/O adres. Register B bevat dan het meest significante deel van het adres (A8 t/m A15). Register "r", een van de zeven generalpurpose registers, wordt geladen met de inhoud van het gespecificeerde adres.

Het maken van een signaal

In het eenvoudigste geval wordt een randapparaat aan- of uitgezet door de computer. Om de toestand van het randapparaat te veranderen, moet het niveau van een logisch signaal veranderen van "0" naar "1" of omgekeerd. Laten we aannemen dat een relais is verbonden met bit 0 van een register "OUT1". Om het relais aan te zetten moet het bit 1 zijn, om het relais uit te schakelen een 0. OUT1 is het adres van het output register. Een programma dat het relais aanzet is:

```
TURNON  LD  A, 00000001B   LAAD PATROON IN A
        OUT (OUT1), A      ZET PATROON IN
                           OUTPUTREGISTER
```

We hebben aangenomen, dat de toestand van de andere zeven bits van A niet belangrijk waren. Dat is echter vaak niet het geval. Ze zouden met andere relais verbonden kunnen zijn. We kunnen het pro-

gramma daarom nog verbeteren. We willen het relais aan zetten, zonder de toestand van de andere bits te veranderen. Aangenomen is, dat de inhoud van het output register kan gelezen worden.

```
TURNON  IN   A, (OUT1)    LEES INHOUD VAN OUT1
        OR   00000001B    BIT 0 WORDT 1
        OUT  (OUT1), A    (OUT1),A
```

Het programma leest eerst de inhoud van adres OUT1. M.b.v. een OR wordt alleen bit 0 veranderd in een 1, de andere bits blijven gelijk. (Dit werd in hoofdstuk 4 behandeld) Zie figuur 6.1.

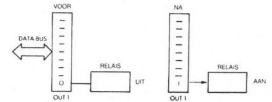


Fig. 6.1: Inschakelen van een relais

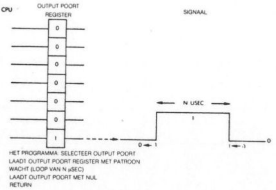


Fig. 6.2: Een geprogrammeerde puls.

Pulsen

Het maken van een puls gebeurt op de zelfde manier als het maken van een niveau, zoals dat hierboven is gedaan. Eerst wordt een output bit 1 gemaakt, daarna weer 0. Het resultaat is een puls. Zie figuur 6.2. Er doet zich echter een probleem voor: hoe maken we een puls van de juiste lengte. We moeten een vertraging inbouwen.

Het opwekken en meten van een vertraging

Een vertraging kan door software en door hardware worden opgewekt. Hier zullen we zien hoe het gedaan kan worden met een programma. Later komen we de hardware oplossing met een "programmeerbare interval timer" (PIT) nog wel tegen.

Een geprogrammeerde vertraging komt tot stand d.m.v. tellen. Een tel register wordt met een bepaalde waarde geladen, waarna er iedere keer 1 van wordt afgetrokken d.m.v. een loop. Het programma blijft in die loop tot dat de teller de waarde 0 heeft. De tijd die hiervoor nodig is, is bepalend voor de vertraging. Bij wijze van voorbeeld volgt hier een programma, dat een vertraging van 69 klok cycli opwekt:

```
DELAY   LD    A, 5       A IS TELLER
NEXT    DEC   A          TREK 1 AF VAN A
        JP    NZ,NEXT    GA NAAR NEXT TOT A = 0
```

Register A wordt met de waarde 5 geladen. De volgende instructie trekt 1 af van A. De daarop volgende instructie test of A nul is; is dat niet het geval, dan wordt naar NEXT gesprongen. Is A wel nul, dan wordt de volgende instructie uitgevoerd. Het programma is simpel en het stroomdiagram is te vinden in figuur 6.3.

We gaan nu de vertraging door het programma opgewekt berekenen.

LDA in de immediate mode: negen cycli, DEC: vier, en tenslotte JP: zeven klok cycli, behalve de laatste keer, dan heeft de instructie 12 cycli nodig.

De totale vertraging is dus: $9 + 5 \times 11 + 5 = 69$ cycli.

Als 1 cyclus 0,5 microseconde duurt, dan is de vertraging 34,5 microseconde.

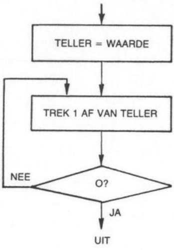


Fig. 6.3: Stroomdiagram basis vertraging

De zoeven beschreven vertraging wordt in de meeste input/output programma's gebruikt. Zorg daarom dat je hem goed begrijpt. Maak de volgende oefeningen:

Oefening 6.1: Wat is de maximale, en wat is de minimale vertraging die met deze drie instructies te maken is?

Oefening 6.2: Verander het programma zodanig, dat een vertraging van ongeveer 100 microseconden ontstaat.

Een eenvoudige methode om de vertraging langer te maken, is het toevoegen van een of meer instructies in het programma tussen DEC en JP. De instructie NOP kan daarvoor gebruikt worden. (NOP doet niets gedurende vier cycli.)

Langere vertragingen

Langere vertragingen d.m.v. software kunnen bereikt worden door de teller groter te maken. Een register paar is te gebruiken als een 16-

bits teller. De teller kan ook drie bytes lang zijn, zoals in het volgende programma:

```
DEL24   LD    B, COUNTH     TELLER HOOG (8 BITS)
DEL16   LD    DE,-1
LOOPA   LD    HL, COUNTL    TELLER LAAG
LOOPB   ADD   HL, DE        VERMINDER TELLER
        JR    C, LOOPB      GA DOOR TOT NUL
        DJNZ  LOOPA         VERMINDER B EN SPRING
```

DE wordt met −1 geladen, en bij HL opgeteld. HL wordt dus met 1 verminderd. B wordt met een waarde geladen, evenals HL. Het programma heeft twee loops. In de binnenste loop wordt TELLER LAAG (= HL) iedere keer met 1 verminderd. Wordt HL 0, dan komen we in de buitenste loop terecht. TELLER HOOG wordt met 1 verminderd, TELLER LAAG wordt opnieuw geladen, en we komen weer in de binnenste loop terecht. Dit gaat door tot TELLER HOOG (= B) de waarde nul heeft. De vertraging is ten einde. Het programma heeft een teller binnen een andere teller. Langere vertragingen zijn te maken, door op dezelfde wijze meer tellers te introduceren.

Het nadeel van deze methode is, dat de processor tijdens deze vertragingen niets anders doet. Als de computer niets anders hoeft te doen, is dat acceptabel. In veel gevallen moet hij echter ook beschikbaar zijn voor andere taken, zodat lange vertragingen meestal niet met software worden gemaakt. Zelfs korte vertragingen kunnen uit den boze zijn, als een bepaalde reactie tijd gegarandeerd moet worden in bepaalde situaties. Als bovendien interrupts zijn toegestaan tijdens een vertragingsloop, is van precisie helemaal geen sprake. Er moeten dan hardware vertragingen gebruikt worden.

Oefening 6.3: Schrijf een programma voor een 100 miliseconde vertraging. (Dit is een typische vertraging van een Teletype.)

Hardware vertragingen

Hardware vertragingen worden verkregen door gebruik te maken van zogenaamde *programmeerbare interval timers*, of timers. Een register van de timer wordt met een waarde geladen. Het verschil is , dat de timer dit register zelf aftelt. Als de timer klaar is, d.w.z. de teller is 0, zendt deze normaal een interrupt naar de processor. Of de timer set een statusbit, dat periodiek door de processor wordt gecontroleerd.

Hoe de interrupts worden gebruikt komt later in dit hoofdstuk aan de orde.

Een timer kan ook andere dingen: tellen vanaf 0, de duur van een signaal tellen, of het aantal ontvangen pulsen tellen. Als de timer werkt als een interval timer, dan wordt gezegt, dat hij werkt in de "*one-shot*" mode. Telt hij pulsen, dan werkt hij in de *puls-tel* mode. Enkele timers bevatten meerdere registers en mogelijkheden, waaruit de programmeur een keuze kan maken.

Het meten van pulsen

Bij het ontvangen en meten van pulsen doet zich een ongeveer gelijk probleem voor als bij het opwekken van pulsen. Bovendien is er nog een ander probleem: het maken van een puls wordt door het programma bestuurd, maar input pulsen komen *asynchroon* met het programma voor. Om een puls te detecteren komen twee methoden in aanmerking: "*polling*" en *interrupts*. Interrupt behandelen we later in dit hoofdstuk.

We bekijken eerst de polling techniek. Bij deze techniek wordt continu de waarde van een input register gelezen, en een bepaald bit wordt daarvan wordt getest. Stel dat dit bit 0 is. Bit 0 is normaal 0. Als een puls wordt ontvangen wordt dit bit 1. Het programma test bit 0 voortdurend, totdat dit 1 wordt, want dan is een puls gedetecteerd. Dat doet het volgende programma:

```
POLL  IN    A, (INPUT)   LEES INPUT REGISTER
ON    BIT   0, A         TEST BIT 0
      JR    Z, POLL      BLIJF TESTEN ALS BIT 0=0
```

Omgekeerd, stel dat de input lijn normaal 1 is, en dat een 0 moet worden getecteerd. Dat is het geval bij een START bit afkomstig van een Teletype. Dan wordt het programma:

```
POLL   IN    A, (INPUT)    LEES INPUT REGISTER
       BIT   0, A          TEST BIT 0
       JR    NZ, POLL      TEST IS OMGEKEERD
START  ...                 T.O.V. HET VORIGE
                           PROGRAMMA
```

Meting van de pulslengte

Het meten van de lengte van een input puls kan op de zelfde manier gebeuren, als het berekenen van de lengte van een output puls. Zowel een software als een hardware techniek is bruikbaar. Als het gebeurt d.m.v. software, wordt een teller regelmatig met 1 verhoogd. Daarna wordt getest of de puls nog aanwezig is. Is de puls er nog, dan springt het programma terug naar het begin van de loop en gaat verder met het tellen. Is de puls niet meer aanwezig, dan is de waarde van de teller een maat voor de lengte van de puls. De lengte kan berekend worden.

```
DURTN    LD    B, 0          TELLER 0
AGAIN    IN    A, (INPUT)    LEES INPUT
         BIT   0, A          TEST BIT 0
         JR    Z, AGAIN      WACHT TOT A = 1
LONGER   INC   B             VERHOOG TELLER
         IN    A, (INPUT)    TEST BIT 0
         BIT   0,A
         JR    NZ, LONGER    WACHT TOT A = 0
```

Natuurlijk nemen we aan, dat bij register B geen overflow optreedt. Kan dat wel gebeuren, dan moet het programma worden verbeterd, anders is dit een programmeer fout!

Aangezien we nu weten hoe pulsen gemaakt en ontvangen moeten worden, kunnen we ons bezig houden met overdrachten van grotere hoeveelheden data, seriele en parallelle data. Daarna zullen we onze kennis toepassen op bestaande randapparatuur.

PARALLELLE WOORD OVERDRACHT

Aanname is, dat 8 bits over te dragen data parallel aanwezig is op adres "INPUT". Zie figuur 6.4. De microprocessor moet deze data lezen, als een status signaal zegt, dat de data geldig zijn. De status informatie staat in bit 7 van adres "STATUS". Het te schrijven programma moet automatisch ieder ontvangen woord lezen en opbergen zo gauw het binnen komt. Het aantal te ontvangen woorden is bekend, en staat op adres "COUNT". Zou het aantal niet bekend zijn, dan moet getest worden op een bepaald karakter, het *break-karakter*. Dat kan zijn *rubout,* of het karakter "*". Daarvoor hebben we reeds een programma gemaakt.

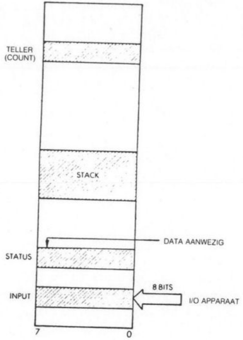


Fig. 6.4: Parallelle woord overdracht

Figuur 6.5 geeft het stroomdiagram. Het status bit wordt getest tot het 1 is, wat betekent dat het woord klaar is. Als een woord klaar is, wordt het gelezen en in het juiste adres geladen. Daarna wordt de teller met 1 verlaagd. Is deze 0 geworden, dan zijn alle data ontvangen. Is dat niet het geval, dan moet het volgende woord worden gelezen. Een simpel programma dat dit algoritme uitvoert volgt nu:

```
PARAL  LD    A, (COUNT)   ZET TELLER IN A
       LD    B, A         B IS TELLER
WATCH  IN    A, (STATUS)  DATA KLAAR? GA DAN
                          DOOR
       BIT   7,A          BIT 7 IS "1" INDIEN DATAKLAAR
       JP    Z, WATCH     TEST STATUS
       IN    A, (INPUT)   LEES DATA
       PUSH  AF           ZET DATA OP STAPEL
```

```
        DEC   B           TREK VAN TELLER 1 AF
        JP    NZ,WATCH    GA NAAR WATCH TOT B=0
```

In het programma wordt aangenomen, dat de "data klaar" vlag automatisch 0 wordt, als de status wordt gelezen. In de praktijk gebeurt dat ook.

De eerste twee registers initialiseren het teller register B:

```
PARAL   LD A,(COUNT)
        LD B,A
```

Er is geen gemakkelijke manier om B vanuit het geheugen te laden. Of we doen het via A, zoals hier, of B en C moeten tegelijk worden geladen.

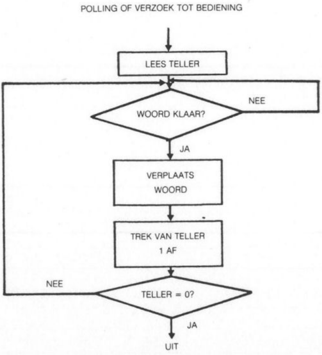


Fig. 6.5: Parallelle woord overdracht

De volgende drie instructies lezen het status bit en veroorzaken een loop zo lang bit 7 is 0. (Dit is het teken bit, bit N.)

```
IN      A,(STATUS)
BIT     7, A          "IN" SET GEEN STATUS BITS
JP      Z,WATCH
```

Als geen sprong wordt uitgevoerd, is de data geldig en kan deze gelezen worden:

```
IN      A,(INPUT)
```

Het woord is nu gelezen en moet opgeborgen worden. Aangenomen dat een voldoende deel van de stapel vrij is, kan dat met de instructie:

```
PUSH    AF
```

Deze instructie plaatst A (en F) op de stapel. Als de stapel vol is, of het aantal over te dragen woorden is groot, kan de stapel niet worden gebruikt. Dan moet het geheugen gebruikt worden. De data kan geplaatst worden m.b.v. bijvoorbeeld een geindexeerde instructie. Dat vereist echter wel een extra instructie om het index register te verhogen of te verlagen. PUSH is sneller (slechts 11 klok cycli).

Het woord is gelezen en weggezet. De woorden teller moet verlaagd worden, en er moet getest worden of we al klaar zijn:

```
DEC     B
JP      NZ,WATCH
```

We gaan door tot de teller 0 is.

Dit negen instructies lange programma kunnen we een "*benchmark*" noemen. Een benchmark is een zorgvuldig geoptimaliseerd programma om de capaciteiten van een gegeven processor onder gegeven omstandigheden te testen. Het programma is ontworpen voor maximale snelheid en efficientie. We zullen nu de maximale overdracht snelheid berekenen van dit programma. Teller staat in het geheugen. Het getal achter iedere instructie die hierop volgt geeft het aantal benodigde klokcycli.

```
PARAL   LD    A, (COUNT)   13
        LD    B, A         4
WATCH   IN    A, (STATUS)  11
        BIT   7, A         8
        JP    Z, WATCH     7/12
        IN    A, (INPUT)   11
        PUSH  AF           11
        DEC   B            4
        JP    NZ, WATCH    7/12
```

De minimale tijd die dit programma nodig heeft is te berekenen door aan te nemen dat iedere keer als STATUS wordt getest, de data aanwezig is. M.a.w. de eerste sprong in het programma vindt niet plaats. De minimale tijd wordt dus:

$$13 + 4 + (11 + 8 + 7 + 11 + 11 + 4 + 7) \times \text{COUNT} + 5$$

Als we de eerste 17 cycli van het initialiseren van het tel register verwaarlozen, komen we op een waarde van 64 klokcycli, of 32 microseconden met een 2MHz klokfrequentie.
De maximale overdracht snelheid is dus:

$$\frac{1}{29.5\ (10^{6})} = 31{,}25 \text{ K bytes per seconde}$$

Oefening 6.4: Neem aan, dat het aantal te verplaatsen woorden groter is dan 256. Verander het programma overeenkomstig, en bereken de maximale overdracht snelheid.

Oefening 6.5: Probeer het programma te verbeteren, zodat de snelheid groter wordt. Gebruik daarbij:
1— JR i.p.v. JP
2— DJNZ
3— INIR i.p.v. INDR.
Was het bovenstaande programma werkelijk optimaal?

We weten nu hoe we parallelle dataoverdrachten met hoge snelheid uit moeten voeren. We kunnen nu overstappen op een meer ingewikkelde soort dataoverdracht.

BIT SERIELE DATAOVERDRACHT

Bij een seriele ingang van een systeem komen de informatie bits na elkaar binnen. Deze bits kunnen een gelijke afstand tot elkaar hebben. Dit wordt gewoonlijk *synchrone* overdracht genoemd. Ook kan de data in reeksen van meerdere bits binnenkomen, waarbij de onderlinge afstand van de reeksen willekeurig is. Dat heet *asynchrone* data overdracht. We zullen een programma ontwerpen, dat op beide manieren kan werken. Het principe voor het ontvangen van seriele data is eenvoudig: we bekijken een ingang, zeg ingang 0. Als een bit wordt gedetecteerd, lezen we de ingang, en schuiven het bit in een buffer. Als op deze manier 8 bits ontvangen zijn, stoppen we het ontvangen byte in het geheugen, en wachten op het volgende. Om het geheel niet te gecompliceerd te maken, nemen we aan, dat het aantal te ontvangen bytes van te voren bekend is. Anders moeten we wachten op een speciaal karakter, dat het einde aangeeft. We weten trouwens al hoe dat moet worden gedaan. In figuur 6.6 staat het stroomdiagramma van het programma. Het programma volgt nu:

```
SERIAL  LD    C, 0          MAAK INPUTWOORD 0
        LD    A, (COUNT)    LAAD B MET BYTE TELLER
        LD    B,A
LOOP    IN    A, (INPUT)    LEES POORT
        BIT   7, A          BIT 7 IS STATUS, BIT 0 DATA
        JR    Z, LOOP       WACHT TOT A = 1
        SRL   A             SCHUIF DATABIT IN CARRY
        RL    C             RED DATABIT IN REG. C
        JR    NC, LOOP      GA DOOR TOT 8 BITS
                            ONTVANGEN ZIJN
        PUSH  BC            ZET WOORD OP STAPEL
        LD    C, 01H        RESET MARKEER BIT
        DEC   B             MAAK BYTE TELLER 1 KLEINER
        JR    NZ, LOOP      HAAL VOLGENDE WOORD
```

Het is een efficient programma, en het maakt gebruik van nieuwe technieken die we zullen uitleggen (zie figuur 6.7).

Er gelden de volgende afspraken: adres TELLER bevat het aantal van de te verplaatsen woorden. Register C wordt gebruikt om het woord uit de ontvangen bits samen te stellen. Adres INPUT heeft betrekking op een input register. Bit 7 daarvan is het status, of klok bit.

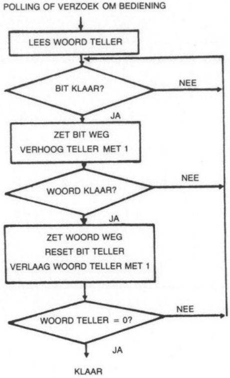


Fig. 6.6: Bit seriele data overdracht. Stroomdiagram.

Als het 0 is, is de data niet geldig, is het 1, dân wel. De data zelf staat op bit 0 van het input register. In veel gevallen zullen status en data ieder hun eigen input register hebben. Het programma is daartoe gemakkelijk te veranderen. Bovendien nemen we aan, dat het eerste te ontvangen bit een 1 is. Is dat niet zo, dan moet een verandering in het programma worden aangebracht, wat we later ook zullen doen. Het programma is exact volgens het stroomdiagram in figuur 6.6 opgebouwd. De eerste paar regels vormen een wachtloop, die test of een bit klaar is. Daarvoor moet het input register ingelezen worden, waarna het zero bit Z getest wordt. Zolang dit bit 0 is, blijven we in de loop. Pas

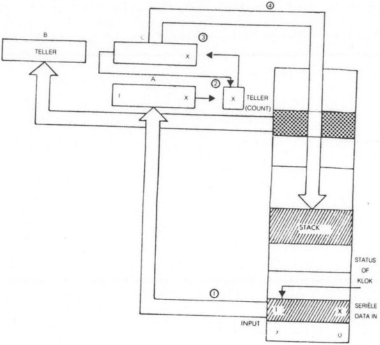


Fig. 6.7: Serie naar parallell. De registers.

als het status (of klok) bit 1 wordt, wordt de volgende instructie uitgevoerd.

Deze serie instructies komt overeen met pijl 1 van figuur 6.7.

De accumulator bevat nu een 1 in bit positie 7, en het data bit in bit positie 0. Het eerste ontvangen bit is een 1, maar de daarop volgende bits kunnen zowel 1 als 0 zijn. Het data bit op positie 0 moet daarom bewaard worden. De instructie:

```
SRL A
```

schuift de inhoud van de accumulator 1 bit naar rechts. Het meest rechtse bit van A, het data bit, valt in het carry bit. Dat bit bewaren we in register C (de pijlen 2 en 3 in figuur 6.7 geven dit proces weer):

```
RL C
```

D.m.v. deze instructie komt het carry bit in het rechtse bit van register C. Tegelijk komt het linker bit van C in het carry bit terecht. (Als je daaraan twijfelt, moet je hoofdstuk 4 nog maar eens nakijken!)

Het is belangrijk dat je onthoudt, dat een roteer instructie het carry bit bewaart, hier in het rechtse bit van C, en dat het oorspronkelijke carry bit in positie 7 van het register komt.

In ons geval wordt het carry bit 0. De volgende instructie:

```
JR  NC, LOOP
```

test de carry en springt terug naar adres LOOP zolang de carry 0 is. Dit is onze automatische bit teller. Na de eerste RL heeft C de waarde "00000001". Het is gemakkelijk in te zien, dat na 8 verschuivingen de 1 tenslotte in de carry terecht komt, waarna de loop eindigt. Het is een ingenieuze manier om een loop teller te maken, zonder er een aparte instructie aan te spenderen. Deze techniek is gebruikt om het programma zo kort mogelijk te maken, en de werking ervan te verbeteren.

Als tenslotte JR NC faalt, dan bevat register C 8 opeenvolgende seriebits. Deze bytes moeten in het geheugen worden opgeslagen. Dat doet de volgende instructie (de pijl 4 van figuur 6.7):

```
PUSH BC
```

De inhouden van B en C worden op de stapel geplaatst. Dat kan alleen als daar genoeg ruimte is. Aangenomen dat dat het geval is, dan is deze methode gewoonlijk de snelste manier om data in het geheugen te zetten. (Zelfs ondanks het feit dat we een onnodig register, B, op de stapel plaatsen). De stackpointer wordt automatisch bijgewerkt. Zouden we de stapel niet gebruiken, dan moest een extra instructie worden toegevoegd, om een pointer naar het geheugen bij te werken. Of we zouden een geïndexeerde instructie kunnen gebruiken, maar ook dan kost het extra tijd om de index te verhogen of te verlagen.

Nadat het eerste woord is ontvangen, bestaat niet langer de garantie dat het eerstvolgende bit een 1 is. Om het register toch als bit teller te kunnen gebruiken, moet het met de waarde "00000001" geladen worden. Dat doet de volgende instructie:

```
LD  C, 01H
```

Omdat een woord is ontvangen, moet de woord teller met 1 worden verminderd. Daarna moet getest worden of we aan het einde van de data overdracht zijn.De volgende twee instructies voeren dit uit:

```
DEC B
JR   NZ, LOOP
```

Het bovenstaande programma is ontworpen voor een hoge snelheid, zodat het een snelle stroom van seriele data kan omtvangen. Is het programma afgelopen, dan is het natuurlijk aan te raden de data van de stapel weg te halen, en ergens anders in het geheugen te plaatsen. In hoofdstuk 2 hebben we al geleerd, hoe een blok data verplaatst kan worden.

Oefening 6.6: Bereken de maximale snelheid, waarmee het programma seriele bits kan ontvangen. Zoek het aantal cycli van iedere instructie op in hoofdstuk 4. Om de lengte in tijd van een loop te berekenen, moet je de duur van een loop, in microseconden, vermenigvuldigen met het aantal keren dat de loop doorlopen wordt. Om de maximale snelheid te berekenen, moet je aannemen, dat de data iedere keer als het status bit wordt getest klaar is.

Het laatste programma is moeilijker te begrijpen dan de voorgaande programma's. Laten we het daarom nog eens gedetailleerd bekijken. (Zie figuur 6.6).

Zo nu en dan wordt een data bit ontvangen in bit 0 van INPUT. Het is mogelijk dat, bijvoorbeeld, achter elkaar drie enen worden ontvangen. Toch moet ieder van deze bits afzonderlijk te herkennen, zijn. Dat is de functie van het klok signaal.

Het klok (of status) signaal vertelt dat de data op een bepaald moment geldig is. Voordat een data bit kan worden gelezen, moet dus eerst het status bit worden getest. Als de status 0 is, moet er worden gewacht, is de status 1, dan is er data binnengekomen.

In ons geval nemen we aan, dat het status bit bit 7 van het register INPUT is.

Oefening 6.7: Kun je uitleggen waarom bit 7 gebruikt wordt voor de status en bit 0 voor de data? Maakt het iets uit?

Als er eenmaal een data bit is ontvangen, moet het op een veilige plaats worden bewaard. Daarna wordt het naar links geschoven, zodat

we klaar zijn voor het volgende bit.

Helaas wordt de accumulator gebruikt voor zowel het lezen als het testen van status en data bit. Daarin kan de data dus niet worden bewaard.

Oefening 6.8: Kan de status worden getest zonder de inhoud van de accumulator te verstoren? Bijvoorbeeld d.m.v. een speciale instructie? Als dat mogelijk is, kan de accumulator dan worden gebruikt om een heel byte in te ontvangen? Kan de snelheid worden verbeterd door een automatische sprong te gebruiken?

Oefening 6.9: Herschrijf het programma, zodat in de accumulator de ontvangen bits worden opgeslagen. Vergelijk dit programma met het vorige m.b.t. snelheid en aantal instructies.

We hebben aangenomen, dat in ons speciale voorbeeld het allereerste bit een 1 is. In het algemeen zal dat niet zo zijn.

Oefening 6.10: Herschrijf het programma, aannemende dat het eerste data bit zowel een 1 als een 0 kan zijn. Hint: de bit teller blijft werken, als deze eerst wordt geinitialiseerd met de juiste waarde.

Tenslotte is de data opgeslagen in de stapel. Dat kan natuurlijk ook in een gespecificeerd stuk geheugen.

Oefening 6.11: Herschrijf het programma en plaats de data in het geheugen, te beginnen bij adres BASE.

Oefening 6.12: Herschrijf het programma zodanig, dat de data overdracht stopt als het karakter "S" wordt ontvangen.

Het hardware alternatief

Zoals gebruikelijk voor de meeste standaard I/O algoritmes is ook deze procedure m.b.v. hardware uit te voeren. De chip die daarvoor wordt gebruikt is de UART. Dit ic ontvangt de bits, en stelt automatisch zelf de bytes samen. Wil men echter het aantal componenten zo klein mogelijk houden, dan moet het programma, of een variant daarop, gebruikt worden.

Oefening 6.13: Herschrijf het programma, daarbij aannemende dat de

data beschikbaar is op positie 0 van adres INPUT, terwijl de status te vinden is op positie 0 van adres INPUT + 1.

SAMENVATTING VAN DE BASIS I/O

We hebben nu geleerd, hoe we elementaire I/O operaties uit moeten voeren. We weten ook hoe een stroom parallelle of seriele data behandeld moet worden. We zijn nu klaar om door te gaan met het volgende onderwerp: de communicatie met echte I/O apparatuur.

COMMUNICATIE MET INPUT/OUTPUT APPARATUUR

Om data met input/output apparatuur uit te kunnen wisselen, moeten we ons er eerst van overtuigen, dat data beschikbaar is, als we willen lezen; of dat het apparaat klaar is om data te ontvangen, als we willen zenden. Daarbij is onderscheid te maken tussen twee procedures: de handshake en de interrupt. Allereerst de handshake.

Handshake

De handshake wordt in het algemeen gebruikt voor de communicatie tussen twee asynchrone apparaten, d.w.z. tussen twee apparaten die niet gesynchroniseerd zijn.

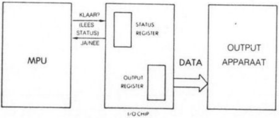


Fig. 6.8: Handshake (output)

Bijvoorbeeld: we willen een woord naar een parallelle printer sturen. Allereerst moeten we zeker weten dat de printer klaar is om te ontvangen. Daarom vragen we de printer:"ben je klaar?" De printer zal met ja of nee antwoorden. Is de printer niet klaar, dan moeten we wachten. Is de printer wel klaar, dan zenden we de data (Zie figuur 6.8).

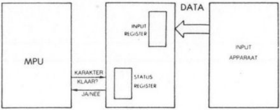


Fig. 6.8a: Handshake (input)

Omgekeerd, als we data willen ontvangen, vragen we of er data aanwezig is. Het randapparaat kan ook hier met ja of nee antwoorden. Dat kan bijvoorbeeld gebeuren d.m.v. status bits (Zie figuur 6.8a).

Deze procedure komen we in het dagelijks leven overal tegen. Als we met iemand willen praten, die misschien met iets anders bezig is, vragen we die persoon of hij (of zij) even tijd heeft. Vaak gaat dat gepaard met handenschudden (handshake). Zijn daarna alle formaliteiten achter de rug, dan kan er gepraat worden.

We zullen deze procedure met een eenvoudig voorbeeld illustreren.

Het zenden van een karakter naar de printer

Het geheugen adres CHAR bevat het te zenden karakter.

```
WAIT    IN    A,(STATUS)
        BIT   7, A          KLAAR?
        JR    Z, WAIT       NEE,GA NAAR WAIT;
                            JA,GA DOOR
        LD    A, (CHAR)     HAAL KARAKTER
        OUT   (PRNTD), A    PRINT KARAKTER
        JR    WAIT          GA DOOR VOOR
                            VOLGENDE KARAKTER
```

Het programma is recht-toe-recht-aan, en gebruikt de hierboven beschreven handshake procedure. Figuur 6.9 laat het data pad zien.

Het karakter (DATA) bevindt zich op adres CHAR. Allereerst wordt de status van de printer getest. Als bit 7 van het status register 1 wordt, betekent dat, dat de printer klaar is. Het karakter wordt dan in de accumulator geladen, en naar de printer verstuurd. Zolang het status bit 0 blijft, blijft het programma in een loop (WAIT) staan wachten.

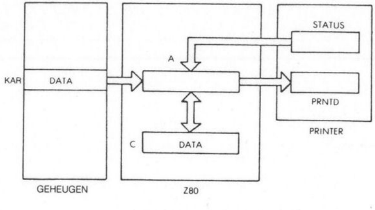


Fig. 6.9: Printer. Data pad.

Oefening 6.14: Hoeveel instructies kunnen we uitsparen in het bovenstaande programma, als het mogelijk zou zijn de data direct in register C te laden, en deze data in C direct naar de printer te sturen?

Oefening 6.15: Bij een echte printer moet meestal eerst een start bevel worden gegeven voordat hij kan worden gebruikt. Verander het programma zodanig, dat zo'n start commando eerst gegeven wordt. Dit commando bestaat uit het 1 maken van bit 0 in het register STATUS.

Oefening 6.16: Als de BIT instructie niet bestond, zou je dan een andere instructie kunnen gebruiken in regel 2 van het programma? Zo ja, geef dan de voordelen van de BIT instructie, als die er zijn, boven de door jou gebruikte instructie.

Oefening 6.17: Verander het programma zodanig, dat een reeks van n karakters wordt geprint. N is kleiner dan 255.

Oefening 6.18: Verander het programma zo, dat een reeks karakters wordt geprint, totdat deze reeks de code voor een "carriage-return" bevat.

De hele output procedure gaan we nu nog eens moeilijker maken, door een code omzetting toe te passen, en door tegelijk naar meerdere randapparaten te zenden.

Output naar een zeven-segments LED

Een gewone zeven-segments licht emmiterende diode (LED) kan de getallen 0 tot en met 9, of de hexadecimale getallen 0 tot en met F weergeven d.m.v. verschillende combinaties van de segmenten. Figuur 6.10 laat een zeven-segments LED zien. De mogelijke getallen staan in figuur 6.11.

De segmenten dragen de namen a tot en met g. (Zie figuur 6.10.) Een 0 bestaat dan uit de segmenten abcdef.

Stel dat bit 0 met een outputpoort verbonden is met segment a, bit 1 met segment b, enz. Bit 7 wordt niet gebruikt. De binaire code die nodig om de segmenten fedcba op te laten lichten is "0111111". Hexadecimaal is dat de code "3F". Maak de volgende oefening.

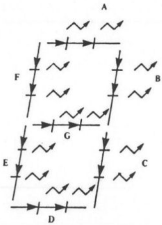


Fig. 6.10: Zeven-segments LED

Oefening 6.19: Bereken de hexadecimale code voor alle hexadecimale getallen op een zeven-segments LED. Vul deze codes in in de volgende tabel.

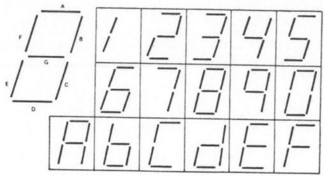


Fig. 6.11: Hexadecimale getallen m.b.v. een zeven-segments LED

Hex	LED code	Hex	LEX code	Hex	LED code	Hex	LED code
0	3F	4		8		C	
1		5		9		D	
2		6		A		E	
3		7		B		F	

De volgende stap is het weergeven van hexadecimale getallen op *meerdere* LED's.

Het aansturen van meerdere LED's

Een LED heeft geen geheugen. Het laat de data alleen zien zolang de segmenten worden geactiveerd. Om de kosten van een LED display zo laag mogelijk te houden, stuurt de microprocessor de LED's een voor een aan. Dat moet zo snel gebeuren, dat de LED's niet zichtbaar gaan knipperen. Laten we een programma maken dat dit doet. Register C zal gebruikt worden om naar de LED te wijzen, die het getal moet weergeven. De accumulator bevat de hexadecimale code van dat getal. Onze eerste stap is de hex code te vertalen in de LED code. In de vorige opgave hebben we daar een tabel voor gemaakt. We kunnen om in die tabel te adresseren geindexeerde adressering toepassen. De verplaatsing is gelijk aan de hexadecimale waarde van het getal. De LED code voor het getal 3 staat dus in het derde element na het basis adres. Het basis adres is SEGBAS. Het programma volgt nu:

```
LEDS    LD    E, A          A BEVAT HEX CODE
        LD    D, 0          DE IS VERPLAATSING
        LD    HL, SEGBAS    HL IS INDEX
        ADD   HL, DE        TABEL ADRES
        LD    A, (HL)       LEES CODE UIT TABEL
        LD    B, 50H        STEL DE VERTRAGING
                            IN
        OUT   (C), A        OUTPUT LED CODE
DELAY   DEC   B             VERTRAGING TELLER
        JR    NZ, DELAY     GA NAAR DELAY ALS B
                            NIET 0 IS
        DEC   C             VOLGENDE LED
        LD    A,C
        CP    MINLED        LAATSTE LED?
        JR    NC,OUT
        LD    BC, (MAXLED)  JA, GEEF C WAARDE VAN
OUT     RET                 EERSTE LED
```

Het programma veromderstelt dat register C het adres bevat van de volgende weer te geven LED, en dat de accumulator het weer te geven getal bevat.

Het programma zoekt eerst de LED code van het getal in A op. Registers D en E vormen de verplaatsing, en H en L een 16-bits index register:

```
LEDS    LD    E, A          BASIS VAN TABEL
        LD    D,0
        LD    HEX CODE
        ADD   HL, DE        TEL BASIS BIJ
                            VERPLAATSING OP
```

Daarna komt de vertragings loop, zodat de LED code een bepaalde tijd wordt weergegeven. Het getal 50H is willekeurig gekozen:

```
        LD    A, (HL)       LEES LED CODE
        LD    B, 50H        VERTRAGING
```

De vertraging komt tot stand d.m.v. een klassieke vertragings loop. De instructie:

```
        OUT   (C),A
```

plaatst de inhoud van A op de output poort waar C naar wijst. (Dat is het LED nummer). De volgende twee instructies vormen de vertragings loop:

```
DELAY   DEC     B
        JR      NZ,DELAY
```

Daarna moet de pointer naar de LED met 1 worden verlaagd. Was het de laatste LED, dan moet de wijzer geladen worden met de waarde van de eerste LED:

```
        DEC     C
        LD      A,C
        CP      MINLED
        JR      NZ,OUT
        LD      BC,(MAXLED)
OUT     RET
```

Het is duidelijk dat het bovenstaande programma geschreven is als subroutine, de laatste instructie is immers een RET instructie.

Oefening 6.20: Normaal is het noodzakelijk de segmenten uit te zetten, voordat naar de volgende LED wordt overgeschakeld. Herschrijf daarvoor het programma. (Voor het nieuwe karakter te verzenden, moet eerst code 00H worden verstuurd.)

oefening 6.21: Wat gebeurt er met het display, als het DELAY label een regel naar boven wordt geschoven? Verandert dat de timing? Ziet het display er anders uit?

Oefening 6.22: De eerste vier regels van het programma voeren in feite een 16-bits geindexeerde adressering uit. Dat gebeurt echter zonder het indexerings mechanisme te gebruiken. Stel dat SEGBAS bij voorbaat bekend is. Noem SEGBSH het hoge deel van het adres, en SEGBSL het lage deel. Plaats SEGBSH in het meest significante deel van het IX register. Herschrijf nu het programma en maak daarbij gebruik van het Z80 indexerings mechanisme. SEGBSL is het verplaatsings veld in de instructie. Wat zijn de voor- en nadelen van dit nieuwe programma?

Oefening 6.23: De registers B, D, E, H en L worden door de subroutine intern gebruikt. De inhoud ervan wordt veranderd. Als de subroutine de geheugen adressen T1, T2, T3, T4 en T5 mag gebruiken, voeg dan aan het begin en het eind van het programma instructies toe, waardoor de inhouden van deze registers bij het verlaten van de routine gelijk zijn aan de inhouden van de registers aan het begin van de routine.

Oefening 6.24: De zelfde opgave als hierboven, maar nu zijn de geheugen adressen T1 enz. niet beschikbaar. (Hint: iedere microprocessor heeft een ingebouwd mechanisme dat informatie in chronologische volgorde kan bewaren.)

We hebben de meeste gewone I/O problemen opgelost. Nu komt een nogal veel voorkomend randapparaat aan bod: de Teletype.

Teletype input-output

De Teletype is een serieel apparaat. Het zendt en ontvangt woorden serieel. Ieder woord is gecodeerd in een 8-bits ASCII code. (In de bijlage is een ASCII tabel opgenomen.) Bovendien wordt ieder woord voorafgegaan door een start bit, en het eindigt met twee stop bits. De toestand van de seriele verbinding is normaal 1. Dit om aan te geven dat de verbinding niet verbroken is. Een start vindt plaats op een overgang van 1 naar 0. Het ontvangende apparaat weet dan dat er data bits volgen. De standaard Teletype verstuurt (en ontvangt) 10 karakters per seconde. Omdat ieder karakter uit 11 bits bestaat, zendt de Teletype 110 bits per seconde. We zullen een programma ontwerpen, dat seriele bits met deze snelheid naar de Teletype stuurt.

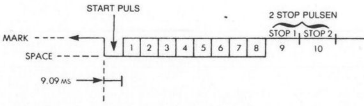


Fig. 6.12: Formaat van een Teletype woord

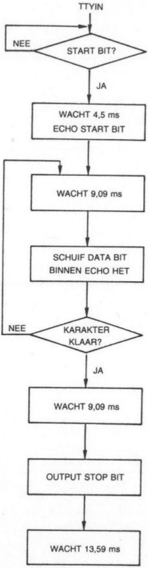


Fig. 6.13: TTY input met echo

110 bits per seconde wil zeggen 1 bit per 9,09 miliseconden. We moeten dus een vertragings loop maken van 9,09 miliseconden. In figuur 6.12 is het formaat van een Teletype woord te zien. Figuur 6.13 geeft het stroomdiagramma voor bit input. Hier is het programma:

```
TTYIN   IN    A, (STATUS)
        BIT   7, A            DATA KLAAR?
        JR    Z, TTYIN        NEE? WACHT
        CALL  DELAY1          JA? WACHT OP MIDDEN
                              VAN PULS
        IN    A, (TTYBIT)     START BIT
        OUT   (TTYBIT), A     ECHO HET
        CALL  DELAY9          VOLGENDE PULS (9 MS)
        LD    B, 08H          BIT TELLER
NEXT    IN    A, (TTYBIT)     LEES DATA BIT
        OUT   (TTYBIT), A     ECHO HET
        SRL   A               REDDEN IN CARRY
        RR    C               PLAATS CARRY IN C
        CALL  DELAY9          VOLGENDE PULS (9 MS)
        DEC   B               TREK 1 AF VAN BIT TELLER
        JR    NZ,NEXT
        IN    A, (TTYBIT)     LEES STOP BIT
        OUT   (TTYBIT), A     ECHO HET
        CALL  DELAY9          SLA TWEEDE STOP BIT OVER
        RET
```

Fig. 6.14: Teletype programma

We bekijken het programma in detail. Allereerst moet de status van de Teletype worden getest, om te kijken of er data aanwezig is:

```
TTYIN   IN    A,(STATUS)
        BIT   7,A
        JR    Z,TTYIN
```

De "BIT" instructie is een zeer bruikbare instructie om een bit in een register mee te testen. Het verandert de inhoud van het te testen register niet. De Z vlag wordt geset als het gespecificeerde bit 0 is, en anders wordt het gereset.

Het programma blijft in de loop tot de status 1 wordt. Het is de klassieke "polling" loop.

Omdat de status niet bewaard hoeft te worden, kan met voordeel

```
        AND   10000000B
```

gebruikt worden in plaats van

```
        BIT    7,A
```

Daardoor worden twee bytes uitgespaard. De inhoud van de accumulator wordt echter vernietigd, maar dat is hier acceptabel.

Onthoud, dat bij het optimaliseren van programma's iedere nieuwe instructie bepaalde neveneffecten kan hebben.

De volgende stap is 4,5 ms te wachten om het midden van de start puls te testen:

```
        CALL  DELAY1
```

DELAY1 is de subroutine die zorgt voor de vertraging van 4,5 ms. Het volgende binnenkomende bit is het start bit. Dit bit moet terug gestuurd worden (echo) naar de Teletype, maar moet verder worden genegeerd:

```
TTYIN   IN    A,(TTYBIT)
        OUT   (TTYBIT),A
```

Voor het eerste data bit moeten we nu 9,09 ms wachten. Dat doet de subroutine DELAY9:

```
        CALL     DELAY9
```

Register B doet dienst als data bit teller en wordt met de waarde 8 geladen:

```
        LD     B,08H
```

Ieder bit wordt gelezen in de accumulator, waarna een echo ervan verstuurd wordt. Bit 0 van de accumulator, waarin het data bit zit, wordt daarna opgeslagen in register C. Daar wordt het in geschoven. De verplaatsing van A naar C gebeurt via het carry bit:

```
NEXT   IN    A,(TTYBIT)
       OUT   (TTYBIT),A
       SRL   A
          C  C
```

Deze volgorde van instructies wordt in figuur 6.15 geillustreerd.

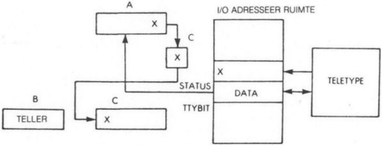


Fig. 6.15: Teletype input

Opnieuw wordt 9 miliseconden gewacht, de bit teller wordt met 1 verlaagd, en zolang niet alle 8 bits binnen zijn wordt de loop doorlopen:

```
CALL  DELAY9
DEC   B
JR    NZ,NEXT
```

Tenslotte wordt het stop bit ontvangen en de echo wordt uitgezonden. Vaak is het zenden van 1 stop bit voldoende, maar m.b.v. twee extra instructies kunnen ook beide worden verzonden:

```
IN    A,(TTYBIT)
OUT   (TTYBIT),A
CALL  DELAY9
RET
```

Bekijk het programma zorgvuldig. De logica ervan is eenvoudig. Het enige nieuwe feit is, dat een ontvangen bit van de Teletype (op adres TTYBIT) terug gestuurd wordt naar de Teletype (echo). Dat is standaard bij een Teletype. Als de gebruiker een toets indrukt, wordt

een karakter naar de processor gestuurd, daarna gaat het karakter weer terug naar het print mechanisme van de Teletype. Het is een bevestiging dat de verbindingen in orde zijn, als het karakter inderdaad juist wordt geprint op het papier.

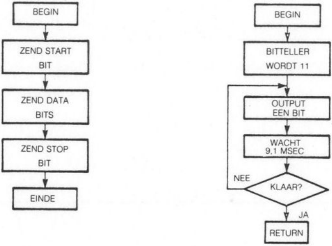


Fig. 6.16: Teletype output

Oefening 6.25: Schrijf een vertragings subroutine voor een vertraging van 9,09 miliseconden (DELAY routine)

Oefening 6.26: Schrijf een PRINTC programma dat de inhoud van geheugen adres CHAR (zie figuur 6.15) print op een Teletype. Neem het bovenstaande programma als voorbeeld.

Hier is het antwoord:

```
PRINTC  LD    B, 11       TELLER = 11 BITS
        LD    A, (CHAR)   HAAL KARAKTER
        OR    A           CARRY = START BIT WORDT 0
        RLA               CARRY IN A
```

```
NEXT    OUT   (TTYBIT), A   OUTPUT
        CALL  DELAY         VERTRAGING
RRA                         VOLGENDE BIT
SCF                         CARRY = 1 (STOP BIT)
DEC           B             BIT TELLER
JR            NZ,NEXT
RET
```

Register B wordt als bit teller gebruikt. De inhoud van bit 0 van register A wordt naar de Teletype gestuurd ("TTYBIT"). Merk op, hoe het carry bit wordt gebruikt om een negende bit te maken (het start bit). Let ook op de manier waarop de carry 0 wordt gemaakt:

```
        OR    A
```

Aan het eind van het programma wordt het carry bit geset (1 gemaakt) d.m.v.:

```
        SCF
```

waardoor het stop bit wordt gevormd.

Oefening 6.27: Verander het programma zodanig, dat gewacht wordt op het START bit i.p.v. het STATUS bit.

Het printen van een reeks karakters

We nemen aan dat de PRINTC routine van oefening 6.26 1 karakter kan printen op onze printer, of beeldbuis, of ieder ander output apparaat. Nu willen we de inhouden van de geheugen adressen START tot START + N printen:

```
PSTRING LD    B, NBR        LENGTE VAN KARAKTER
                            REEKS
        LD    HL, START     BASIS ADRES
NEXT    LD    A, (HL)       HAAL KARAKTER
        CALL  PRINTC        PRINT HET KARAKTER
        INC   HL            VOLGENDE ELEMENT
        DEC   B
        JR    NZ, NEXT      DOE HET NOG EENS
        RET
```

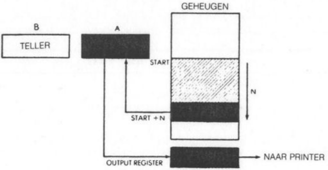


Fig. 6.17: Het printen van een reeks karakters

SAMENVATTING RANDAPPARATUUR

De basis technieken voor het communiceren met typische I/O apparatuur is nu besproken. Buiten het zenden van data is het vaak nodig een of meerdere controle registers te laden met de juiste waarden, waardoor transport snelheid, interrupt mechanisme, en nog vele andere parameters worden ingesteld. Dat moet gebeuren aan de hand van de handboeken van ieder apparaat. (Meer informatie over dit onderwerp is te vinden in een ander Sybex boek: Microprocessor Interfacing Techniques.)

We kunnen nu met ieder apparaat apart communiceren. In een echt systeem zijn alle randapparaten met de bussen verbonden, en kunnen tegelijk om aandacht vragen. Hoe dat aangepakt moet worden leren we in het volgende deel.

INPUT/OUTPUT SCHEDULING

Aangezien verzoeken van I/O apparatuur tegelijk kunnen binnen komen bij de processor, is een soort dienstregeling nodig om te bepalen wie het eerst komt, enz. Dat wordt "scheduling" genoemd. Drie basis I/O technieken worden daarbij gebruikt, die gecombineerd kunnen worden. Deze technieken zijn: polling, interrupt, en DMA. Polling en interrupts zullen we hier behandelen. DMA is een hardware techniek, en zal daarom hier niet aan de orde komen. (Wel in de Sybex boeken From chips to systems: an introduction to microprocessors en Microprocessor Interfacing Techniques.)

Polling

Principieel is polling de eenvoudigste manier om meerdere randapparaten te bedienen. De processor vraagt bij deze methode alle randapparaten verbonden met de bussen om de beurt, of deze iets te vertellen hebben. Wil een randapparaat communiceren met de microprocessor, dan wordt dat toegestaan. Wil het randapparaat niet communiceren, dan gaat de processor verder met het volgende randapparaat, enz. Polling wordt niet alleen toegepast bij de randapparaten, maar ook bij de I/O subroutines.

Bijvoorbeeld: een systeem is uitgerust met een Teletype, een bandopnemer en een beeldscherm. Allereerst vraagt de processor aan de Teletype: "wil je een karakter zenden?". Is het antwoord nee, dan gaat de processor verder. Deze vraagt dan aan de Teletype *output routine*: "wil je een karakter naar de Teletype sturen?". Op deze wijze komen alle randapparaten en in aanmerking komende routines aan de beurt. De figuren 6.20 en 6.21 geven stroomdiagrammen van voorbeelden voor het lezen van een ponsband lezer en voor het printen op een printer.

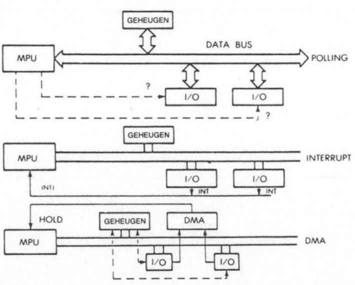


Fig. 6.18: Drie methoden voor I/O controle

Voorbeeld: een polling loop voor de randapparaten 1, 2, 3, en 4 (zie figuur 6.19):

```
POLL4 IN    A, (STATUS 1) HAAL STATUS VAN APPARAAT 1
      BIT   7, A          WIL HET BEDIEND WORDEN?
      CALL  NZ, ONE       JA? GA NAAR SUBROUTINE
      IN    A, (STATUS2)  RANDAPPARAAT 2
      BIT   7,A
      CALL  NZ,TWO
      IN    A, (STATUS3)  RANDAPPARAAT 3
      BIT   7,A
      CALL  NZ,THREE
      IN    A, (STATUS4)  RANDAPPARAAT 4
      BIT   7,A
      CALL  NZ, FOUR
      JR    POLL4         BEGIN OPNIEUW
```

Bit 7 van het status register is 1 voor ieder randapparaat als het wil communiceren met de processor. Wordt een verzoek daartoe waargenomen, dan springt het programma naar de I/O subroutine voor het desbetreffende apparaat.

Over een detail in het programma kunnen we nog iets zeggen. Het is belangrijk te weten hoe iedere instructie de vlaggen beinvloedt. De input instructie verandert niets aan de vlaggen. Zou een LD instructie zijn gebruikt, dan zou bit 7 terug te vinden zijn in het tekenbit. De instructie BIT 7,A zou daarmee overbodig zijn geworden.

Bij sommige toepassingen moeten randapparaten behandeld worden als waren het adressen in het geheugen. Deze techniek wordt "memory-mapped input/output" genoemd. In dat geval moet de IN instructie in het voorbeeld vervangen worden door LD. De rest van het programma, zoals dat hierboven is besproken, moet daaraan aangepast worden.

De voordelen van polling zijn duidelijk: het is simpel, geen hardware hulpmiddelen zijn nodig, en alle input/output loopt synchroon met het programma. Het nadeel is al even duidelijk: een groot deel van de beschikbare tijd wordt verspild aan randapparaten, die niet willen communiceren. Doordat zoveel tijd wordt verspild, is de processor soms te laat voor een ander apparaat.

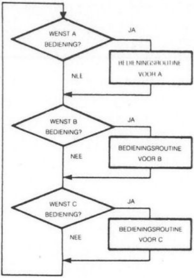


Fig. 6.19: Stroomdiagram polling loop

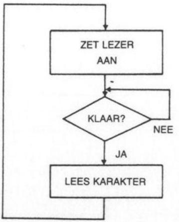


Fig. 6.20: Lezen van een ponsbandlezer

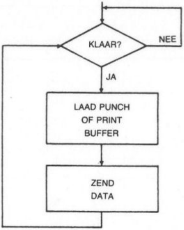


Fig. 6.21: Printen op een ponser of printer

We moeten daarom zoeken naar een ander mechanisme, waarbij de processor de tijd efficienter besteedt. Polling is echter uitstekend geschikt in diè toepassingen, waar de microprocessor toch niets anders heeft te doen. De gehele organisatie blijft daardoor eenvoudig. Maar nu het alternatief voor polling: interrupts.

Interrupts

Het grondbeginsel van interrupts is geillustreerd in figuur 6.18. Een hardware verbinding, de interrupt lijn, is verbonden met een speciale pen van de microprocessor. Meerdere randapparaten kunnen met deze pen worden verbonden. Als een van de apparaten bediend wil worden, wordt een signaal of een puls over de verbinding gestuurd. Een interrupt signaal is een verzoek van een randapparaat aan de processor om bediend te worden. Hoe reageert de microprocessor op de interrupt?

In ieder geval maakt de processor de instructie waar deze mee bezig is af, anders zou er een chaos ontstaan. Daarna springt de microprocessor naar de routine die de interrupt afhandelt. Dit houdt in dat de inhoud van de programmateller geredt moet worden op de stapel. *Een interrupt moet dus ervoor zorgen, dat de inhoud van de programmateller automatisch gered wordt op de stapel.* Omdat de inhoud van het

vlaggen of status register F veranderd wordt door de subroutine, moet deze inhoud ook op de stapel worden gezet. Als de routine bovendien een intern register verandert, moet ook daarvan de inhoud worden gered. (Zie de figuren 6.22 en 6.23).

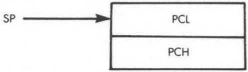


Fig. 6.22: Z80 stapel na de interrupt

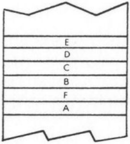


Fig. 6.23: Het redden van enkele werkregisters

Zijn al deze registers goed opgeborgen, dan pas kan naar de interrupt routine worden gesprongen. Aan het eind van deze routine moeten alle oorspronkelijke register inhouden weer worden geladen. Na een speciale interrupt return wordt het hoofdprogramma verder uitgevoerd. We gaan de speciale interrupt lijnen van de Z80 eens nader bekijken.

Z80 interrupts

Een interrupt is een signaal naar de processor, dat verzoekt om bediening. Dat signaal kan ieder moment optreden, en is asynchroon met het programma. Dit in tegenstelling tot een sprong naar een subroutine. Zo'n sprong wordt uitgevoerd door het programma, en is dientengevolge *synchroon* daarmee. Een interrupt stopt in het algemeen het hoofdprogramma (zonder dat deze dat weet).

De Z80 is voorzien van drie interrupt mechanismen: de "busrequest" (BUSRQ), de niet-maskeerbare interrupt (NMI), en de gewone interrupt (INT).

De busrequest

De busrequest (een request is een verzoek) is het interrupt mechanisme met de hoogste prioriteit bij de Z80. De opeenvolging van handelingen wordt in figuur 6.24 getoond. In het algemeen geldt, dat een interrupt pas wordt herkend door de Z80 als de huidige machine cyclus

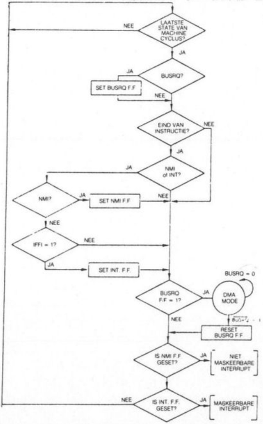


Fig. 6.24: Opeenvolging van handelingen bij een interrupt

afgemaakt is. De NMI en de INT worden pas uitgevoerd als de instructie klaar is. De BUSRQ echter wordt al uitgevoerd nadat de machine cyclus voltooid is, zonder dat op het einde van de instructie gewacht hoeft te worden. Het wordt voor directe geheugen toegang, DMA (Direct Memory Access), gebruikt, en dwingt de Z80 in de DMA mode. (In het Sybex boek "From Chips to Systems" wordt het DMA mechanisme verklaard).

NMI en INT worden onthouden totdat de instructie afgelopen is, d.m.v. het setten van flip-flops: de NMI flip-flop en de INT flip-flop. In de DMA mode houdt de Z80 op met het uitvoeren van instructies en maakt de data en adres bus vrij (hoge impedantie). DMA wordt meestal gebruikt voor data overdrachten met hoge snelheid tussen een snel randapparaat en het geheugen, waarbij gebruikt wordt gemaakt van de data en adres bus. Het einde van een DMA merkt de Z80 op door een verandering van de toestand van de BUSRQ lijn. De processor begint weer normaal te werken. Allereerst worden de NMI en INT flip-flops getest. Is een van deze flip-flops geset, dan wordt de interrupt uitgevoerd.

De programmeur hoeft zich normaal niet met DMA bezig te houden, tenzij snelheid belangrijk is. Heeft het microprocessor systeem een "DMA controller", dan moet de programmeur zich ervan bewust zijn, dat DMA de reactie tijd op een NMI en een INT zeer kan vertragen.

De niet-maskeerbare interrupt

De programmeur kan dit soort interrupt niet verbieden. Vandaar de naam niet-maskeerbaar. Deze interrupt wordt, na voltooiing van de instructie, altijd door de Z80 geaccepteerd. (Komt deze interrupt tijdens een DMA, dan wordt NMI na het eind van BUSRQ geaccepteerd.)

NMI veroorzaakt een automatische push van de programmateller op de stapel, en een sprong naar een 16-bits *interrupt vector* op adres 0066H. Een 16-bits adres op adres 0066H wordt in de programmateller geladen. Dit adres is het start adres van de interrupt routine van de NMI (Zie figuur 6.25).

Deze interrupt heeft een hoge snelheid, en wordt gebruikt voor "noodgevallen". NMI is daarom niet zo flexibel als de maskeerbare interrupt, die hierna beschreven wordt.

Opgemerkt moet nog worden, dat de interrupt vector op adres 066H geladen moet worden vòòr de NMI gebruikt wordt.

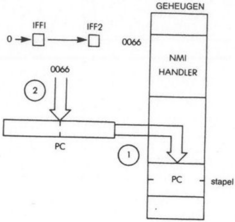


Fig. 6.25: NMI interrupt vector

Een "interrupt vector" is eenvoudig een adres of pointer, welke de locatie van de interrupt routine aangeeft. In dit geval is het een vorm van *indirecte geheugen adressering*. De hardware springt altijd naar adres 0066H, maar de programmeur kan op dat adres ieder mogelijk routine adres plaatsen.

PC	⟶	STACK	(red programma teller)
IFF1	⟶	IFF2	(red IFF)
0	⟶	IFF1	(reset IFF)
JUMP TO 0066H			(roep interrupt routine op)

De status van de interrupt-masker-bit flip-flop (IFF1) wordt ten tijde van een NMI automatisch opgeborgen in IFF2. Daarna wordt IFF1 gereset om eventuele volgende interrupts te voorkomen. Dit voorkomt het verlies van INT's met een lagere prioriteit, en maakt de externe hardware eenvoudiger: de status van een nog niet verwerkte INT wordt intern in de Z80 opgeslagen.

De NMI wordt normaal gebruikt voor gebeurtenissen met een hoge prioriteit, zoals een klok, of bij het uitvallen van de voeding.

IFF2	⟶	IFF1	(herstel IFF)
STACK	⟶	PC	(herstel programma teller)

De terugkeer uit een NMI routine gebeurt met een speciale instructie, RETN: "return uit een niet-maskeerbare interrupt". De inhoud van IFF2 gaat naar IFF1, en de programma teller krijgt zijn oude waarde van de stapel terug. Omdat tijdens de interrupt IFF1 gereset is, kunnen geen andere INT's worden geaccepteerd gedurende NMI: er is geen verlies van informatie.

Interrupt

De gewone, maskeerbare, interrupt kan op drie manieren werken. Dat geldt alleen voor de Z80. De 8080 heeft slechts één soort interrupt. INT kan selectief door de programmeur worden gemaskeerd. Door IFF1 en IFF2 1 te maken, worden interrupts toegestaan. Worden beide flip-flops 0 gemaakt, dan worden interrupts niet waargenomen. De instructie EI set de flip-flops, door DI worden ze gereset. De flip-flops worden gelijktijdig geset of gereset. Tijdens de instructies worden INT's niet geaccepteerd, om het verlies van informatie te voorkomen.

Laten we nu de drie verschillende interrupt modes eens bestuderen:

Interrupt mode 0

Deze mode is identiek aan de 8080 interrupt mode. De Z80 werkt in interrupt mode 0 als deze net gestart is (na een RESET signaal), of als een IM0 instructie is uitgevoerd. Eenmaal in deze mode, wordt een interrupt herkend als IFF1 geset is, mits geen bus-request of niet-maskeesbare interrupt optreedt op het zelfde moment. De interrupt wordt gedetecteerd aan het eind van een instructie. De Z80 reageert op de interrupt door IORQ op te wekken (en een M1 signaal), maar doet verder niets, behalve wachten.

Het is de verantwoordelikheid van het externe randapparaat de signalen IORQ en M1 (dit wordt de interrupt bevestiging of INTA genoemd) te herkennen, en een instructie op de data bus te plaatsen. De Z80 verwacht deze instructie van het randapparaat binnen de volgende klok cyclus. Meestal wordt een RST of een CALL op de bus geplaatst. Beide instructies bewaren de programma teller op de stapel en springen naar een bepaald adres. Het voordeel van RST is, dat het maar een byte nodig heeft, d.w.z. de instructie is snel uit te voeren. Het nadeel is, dat slechts naar 8 adressen op pagina 0 (adressen 0 tot 225) gesprongen kan worden. Het voordeel van CALL is, dat een volledig 16-bits adres meegeven wordt. De instructie is echter drie bytes groot en is dus niet zo snel.

Opgemerkt moet worden, dat als eenmaal een interrupt door de processor wordt verwerkt, geen andere interrupt wordt geaccepteerd. IFF1 en IFF2 zijn dan automatisch 0 gemaakt. Het is de verantwoordelijkheid van de programmeur een EI instructie toe te voegen op de juiste plaats in het programma, in ieder geval voor de terugkeer uit de interrupt, als hij verdere interrupts wil toestaan.

Figuur 6.26 geeft een uitgebreid stroomdiagram betreffende de opeenvolgende handelingen van de mode 0 interrupts.

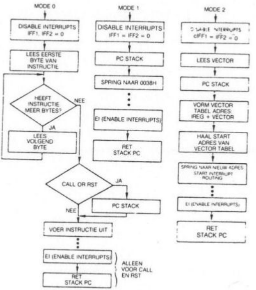


Fig. 6.26: Interrupt modes

De terugkeer uit een interrupt gebeurt m.b.v. een RETI instructie. Op dat moment moet de programmeur altijd de IFF vlaggen op de juiste waarde setten. Deze taak kan echter ook door een randapparaat controler worden overgenomen.

Is het mogelijk dat de interrupt routine de inhoud van enig intern register verandert, dan is de programmeur ook verantwoordelijk voor het op de stapel zetten van deze registers, voordat naar de interrupt

routine gesprongen wordt. Wordt dat niet gedaan, dan zal de inhoud vernietigd worden, en bij terugkeer uit de routine zal het hoofdprogramma fout gaan. Worden bijvoorbeeld de registers A, B, C, D, E, H en L in de interrupt routine gebruikt, dan moeten ze op de stapel worden geplaatst (Zie figuur 6.27).

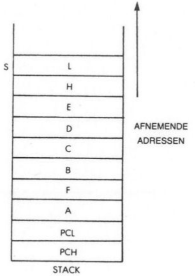


Fig. 6.27: Het redden van de registers

Het overeenkomstige programma is:

```
SAVREG  PUSH AF
        PUSH BC
        PUSH DE
        PUSH HL
```

Aan het eind van de routine moeten de oude inhouden weer in de registers geladen worden. De interrupt routine moet eindigen met de volgende instructies:

```
POP  HL
POP  DE
POP  BC
POP  AF
EI              (tenzij EI al eerder in
                de routine is gebruikt)
```

Dit moet bovendien met de registers IV en IY gedaan worden als ze in de interrupt routine worden gebruikt.

Interrupt mode 1

Deze mode wordt gebruikt na de instructie IM1. Er wordt automatisch gesprongen naar adres 0038H. Het lijkt daarom erg op de NMI interrupt. Het verschil is, dat deze interrupt wel gemaskeerd kan worden. De Z80 bewaart automatisch de inhoud van PC op de stapel (Zie figuur 6.28).

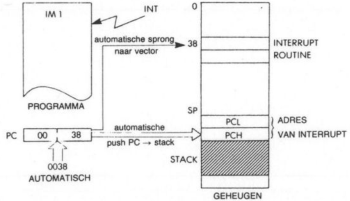


Fig. 6.28: Mode 1 interrupt

Deze reactie op een interrupt (springen naar adres 38H) minimaliseert de benodigde hoeveelheid hardware bij interrupts. Het nadeel is, dat altijd naar dezelfde plaats gesprongen wordt. Als meerdere apparaten een interrupt kunnen veroorzaken, dan moet de routine op adres

38H uitzoeken van welk randapparaat de interrupt afkomstig is. Dat probleem zal hierna nog worden behandeld.

Er moet een voorzorg worden genomen m.b.t. de timing van het interrupt signaal: tijdens in- en uitvoer negeert de Z80 alle data op de data bus tijdens de klokcyclus volgend op de interrupt.

Interrupt mode 2 ("vectored interrupts")

De Z80 komt in deze mode d.m.v de instructie IM2. Het is een krachtige interrupt d.m.v vectoren. Het randapparaat geeft niet alleen een interrupt signaal, maar ook een vector door aan de Z80. De vector vormt een pointer naar het startadres van de interrupt routine. Het is een soort indirecte adresserings methode. Ieder randapparaat vormt een 7-bits adres, dat toegevoegd wordt aan het adres in register I in de Z80. Het rechtse bit van het uiteindelijke adres wordt 0. Dit adres is het begin van een tabel ergens in het geheugen. Ieder 16-bits dubbel woord in de tabel is een adres van een interrupt routine. Dit wordt in figuur 6.29 en 6.30 geillustreerd.

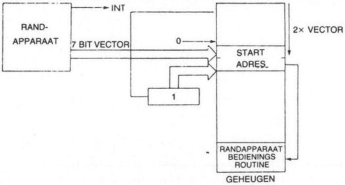


Fig. 6.29: Mode 2 interrupt

De interrupt tabel kan tot 128 ingangen hebben. In deze mode plaatst de Z80 ook automatisch de programma teller op de stapel. Dat is nodig, omdat PC geladen wordt met de inhoud van de interrupt tabel overeenkomstig de vector afkomstig van het randapparaat.

Bijkomend probleem bij interrupts: de "overhead"

Figuur 6.18 geeft in een tekening het verschil tussen polling en interrupts. Hier is te zien, dat tijdens polling veel tijd verloren gaat.

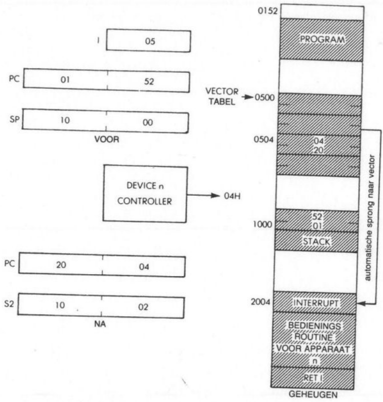


Fig. 6.30: Mode 2: een praktisch voorbeeld

Bij interrupts wordt het programma onderbroken, de interrupt wordt uitgevoerd, en het programma gaat door. Het duidelijke nadeel van een interrupt is de introductie van enkele instructies aan het begin en aan het eind van de interrupt routine (de zogenaamde interrupt overhead), waardoor een vertraging ontstaat.

Oefening 6.28: Bereken, door het tellen van het aantal cycli, hoeveel tijd verloren gaat bij het redden en later weer laden van de registers A, B, D en H.

Nu we weten hoe een interrupt werkt, blijven twee belangrijke problemen over:

1 - Wat te doen als meerdere randapparaten tegelijk een interrupt willen geven?
2 - Wat moeten we doen als tijdens een interrupt een tweede interrupt voorkomt?

Meerdere randapparaten verbonden met 1 interrupt lijn

Door een interrupt springt de processor naar een bepaald adres. Voordat het de interrupt kan afhandelen, moet eerst worden uitgezocht van welk apparaat de interrupt afkomstig is. Zoals gewoonlijk zijn ook hier twee methoden beschikbaar: een software en een hardware methode.

In de software methode wordt polling gebruikt: de microprocessor vraagt ieder randapparaat om de beurt:"is de interrupt van jou afkomstig?". Is het antwoord negatief, dan wordt het volgende apparaat hetzelfde gevraagd. Dit proces is in figuur 6.31 geïllustreerd. Een voorbeeld van een dergelijk programma is:

```
POLINT  IN    A, (STATUS1)  LEES STATUS
        BIT   7, A          VROEG APPARAAT
                            INTERRUPT?
        JP    NZ, ONE       JA,SPRING NAAR ROUTINE
        IN    A, (STATUS2)  NEE, GA DOOR MET
                            VOLGEND APP.
        BIT   7,A
        JP    NZ,TWO
        etc.  ---
```

De hardware methode gebruikt meer componenten in de schakeling, maar geeft tegelijk met de interrupt het adres van het randapparaat dat de interrupt veroorzaakt. De chip die daar tegenwoordig algemeen voor wordt gebruikt is de "PIC", de prioriteit-interrupt-controler. Zo'n PIC plaatst het sprong adres automatisch op de data bus.

Om iets precieser te zijn: in mode 0 plaatst de PIC een 1-byte RST of

3-bytes CALL op de data bus als reactie op de bevestiging van de processor op de interrupt.

NB: een subroutine call is noodzakelijk, omdat de Z80 in mode 0 PC niet redt.

In de meeste gevallen is de reactie tijd op de interrupt niet zo belangrijk, en kan polling gebruikt worden. De hardware methode moet gebruikt worden, als snelheid een van de vereisten is.

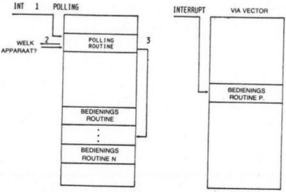


Fig. 6.31: Polling en Vectored interrupts

Simultane interrupts

Het volgende probleem dat kan voorkomen is, dat een nieuwe interrupt kan komen tijdens het uitvoeren van een interrupt routine. Laten we eens gaan kijken wat er gebeurt en hoe de stapel gebruikt kan worden om dit probleem op te lossen. In hoofdstuk 2 is al aangegeven, dat de stapel hierbij een belangrijke rol kan spelen. Het is nu tijd om dat aan te tonen. Aan de hand van figuur 6.33 wordt meervoudige interrupts geillustreerd. De tijdas loopt van links naar rechts in de tekening. De inhoud van de stapel wordt beneden in de tekening getoond. Op tijdstip T0 wordt het programma P uitgevoerd (links in de tekening). We gaan verder in de tijd, naar rechts in de tekening. Op T1 verschijnt interrupt I1. We nemen aan, dat de interrupt-flip-flops zo staan, dat de interrupt is toegestaan. Programma P wordt onderbroken en voorlopig uitgesteld. Dat wordt onder in de tekening getoond. De stapel bevat de programma teller en het status register van P, plus eventueel enkele interne registers, die door de interrupt routine gered zijn.



Fig. 6.32: Meerdere randapparaten kunnen de zelfde interrupt lijn gebruiken

Van tijdstip T1 tot T2 wordt I1 uitgevoerd. Op T2 verschijnt interrupt I2, met een hogere prioriteit dan I1. Zou I2 een lagere prioriteit hebben dan I1, dan zou hij genegeerd worden tot na het beeindigen van I1. Op T2 worden alle voor I1 belangrijke registers op de stappel gezet. Opnieuw komen PC en AF op de stapel. De routine voor I2 kan ook nog besluiten enkele andere registers op de stapel te zetten. I2 wordt helemaal afgemaakt en eindigt op tijdstip T3. Dan wordt de inhoud van de stapel weer in de goede registers geladen. Zie onder in de tekening.

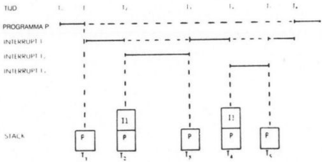


Fig. 6.33: Stapel tijdens meervoudige interrupts

Dat gebeurt automatisch. Het gevolg is, dat I1 verder wordt uitgevoerd. Helaas verschijnt interrupt I3 met een hogere prioriteit op tijdstip T4. Opnieuw worden de registers voor I1 op de stapel gezet. I3 eindigt op T5. De registers voor I1 worden weer geladen en I1 wordt verder uitgevoerd. Ditmaal wordt de routine voor I1 afgewerkt, en eindigt op tijdstip T6. De overgebleven op de stapel gezette registers worden geladen, en programma P kan verder worden uitgevoerd. De lezer kan zelf controleren, dat de stapel nu leeg is.

Oefening 6.29: Neem aan, dat de beschikbare ruimte op de stapel beperkt is tot 300 locaties in een specifiek programma. Neem bovendien aan, dat alle registers altijd gered moeten worden, en dat geneste interrupts (interrupts in interrupts) toegestaan zijn. Hoeveel meervoudige interrupts kunnen worden verwerkt? Zijn er andere factoren, waardoor dit aantal gereduceerd kan worden?

Er moet op gewezen worden, dat in de praktijk slechts een beperkt aantal randapparaten met een systeem verbonden is. Daarom is het onwaarschijnlijk, dat een groot aantal meervoudige interrupts in het systeem voorkomt.

Alle problemen m.b.t. interrupts hebben we nu opgelost. Het gebruik van interrupts is eenvoudig, en ze kunnen met voordeel worden toegepast, zelfs door de beginnende programmeur.

SAMENVATTING

In dit hoofdstuk hebben we gesproken over de technieken, waarmee met de buitenwereld gecommuniceerd kan worden. Van elementaire input/output routines tot complexe programma's voor communicatie met echte randapparatuur. We kunnen de gebruikelijke programma's daarvoor maken. Bij de parallelle data overdracht en de parallel-naar-serie omzetting hebben we zelfs de efficientie van benchmark programma's onderzocht. (Een benchmark programma is een zo efficient mogelijk programma voor het testen van een microprocessor). Tenslotte zijn de polling en interrupt technieken aan de orde gekomen. Natuurlijk kunnen vele andere exotische randapparaten met het systeem verbonden worden. Met de hier behandelde technieken, en door deze randapparaten goed te begrijpen moet het mogelijk zijn alle daarmee samenhangende problemen op te lossen.

In het volgende hoofdstuk worden de interface chips, die in een Z80 systeem gewoonlijk worden gebruikt, behandeld. Daarna zullen basis data structuren, die de programmeur kan gebruiken, aan de orde komen.

Oefening 6.30: Bereken de overhead in mode 0, wanneer alle registers gered moeten worden, en een RST is ontvangen als reactie op de interrupt bevestiging. De overhead wordt gedefinieerd als de totale vertraging, zonder de instructies die de echte interrupt verwerken.

Oefening 6.31: Een 7-segments LED display kan behalve de hex getallen ook andere karakters weergeven. Bereken de codes voor de volgende karakters: H, I, J, L, O, P, S, U, Y, g, h, i, j, l, n, o, p, r, t, u, en y.

Oefening 6.32: Figuur 6.34 geeft een stroomdiagram van het beheer van interrupts. Beantwoord de volgende vragen:

a - wat wordt door de hardware, en wat door de software gedaan?
b - wat is het nut van een masker?
c - hoeveel registers moeten worden bewaard?
d - hoe wordt een apparaat, dat een interrupt geeft geidentificeerd?
e - wat doet de RETI instructie? Wat is het verschil met een terugkeer uit een subroutine?
f - geef een suggestie hoe een stapel overflow behandeld kan worden.
g - wat is de overhead (verloren tijd) geintroduceerd door een interrupt mechanisme?

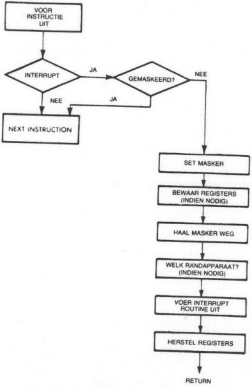


Fig. 6.34: Interrupt logica

7
INPUT/OUTPUT APPARATEN

INLEIDING

We hebben geleerd, hoe de Z80 geprogrammeerd moet worden in de meeste gewone situaties. Speciale aandacht verdienen echter nog de input/output chips, die dankzij nieuwe technieken (LSI) mogelijk zijn geworden. Het gevolg van de introductie van deze chips is, dat niet alleen de microprocessor geprogrammeerd moet worden, maar ook deze nieuwe I/O chips. In feite is het vaak moeilijker te onthouden hoe alle I/O chips geprogrammeerd moeten worden, dan de processor zelf! Niet omdat het programmeren zelf zo moeilijk is, maar vanwege het feit, dat iedere chip zijn eigen eigenaardigheden heeft. We gaan de meest algemene I/O chip bekijken, de parallelle input/output chip (of afgekort "PIO"), en enkele Zilog I/O chips.

DE "STANDAARD PIO"

Er is geen standaard PIO. De meeste PIO'S van de verschillende fabrikanten zijn echter functioneel in principe gelijk. Een PIO is in hoofdzaak een meervoudige poort voor de verbindingen met randapparaten. (Een poort is een verzameling van 8 verbindingslijnen met de buitenwereld). Iedere PIO heeft tenminste twee van dergelijke poorten. Iedere PIO heeft tenminste een *data buffer* nodig, om de inhoud van de data bus te stabiliseren als de data naar buiten wordt gestuurd. Onze PIO is daarom tenminste voorzien van een buffer voor iedere poort.

Bovendien hebben we al gezien, dat de microprocessor een *handshake* procedure, of anders een *interrupt* gebruikt om met het I/O apparaat te communiceren. De PIO zal een dergelijke procedure gebruiken voor de communicatie met het randapparaat. Iedere PIO moet dus voorzien zijn van tenminste *twee controle lijnen per poort* voor de handshake procedure.

De microprocessor moet ook in staat zijn de status van het randapparaat te lezen. Iedere poort moet dus voorzien zijn van een of meer status bits.

Tenslotte kent iedere PIO meerdere opties. De programmeur kan programmeren van welke optie gebruik wordt gemaakt, door het laden van een speciaal register in de PIO. Dit speciale register is het *controle-register*. Soms is de status informatie onderdeel van het controle-register.

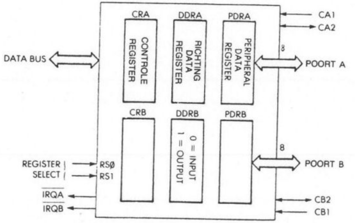


Fig. 7.1: Een typische PIO

Een essentieel kenmerk van de PIO is het feit, dat iedere lijn zowel als input als output gebruikt kan worden. Figuur 7.1 geeft een blokschema van een PIO. De programmeur kan specificeren of een lijn als input of als uitput gebruikt gaat worden. Daarvoor heeft iedere poort een *data-richting register*. Een 0 in een bit van het data-richting register betekent input, een 1 betekent output.

De waarden 0 en 1 voor input en output zijn met opzet zo gekozen.

Stel dat het andersom zou zijn, en een PIO is met een gevaarlijk randapparaat verbonden. Als de processor wordt ingeschakeld, worden meestal alle registers gereset, ze worden dus 0. De PIO is ingesteld als een output poort. Het gevaarlijke randapparaat kan per ongeluk in werking komen. Om zo'n situatie te vermijden, moet de PIO als input geschakeld zijn!

De verbindingen met de microprocessor staan links in figuur 7.1. De PIO is natuurlijk met de data bus verbonden, de adres bus en de controle bus. De programmeur kan eenvoudig het adres van ieder register specificeren, om toegang tot de PIO te krijgen.

Het interne controle register

Het controle register van de PIO voorziet in een aantal opties voor het opwekken van interrupts en handshake procedures. Een volledige beschrijving is hier uiteraard niet nodig. De lezer die een PIO in zijn systeem wil gebruiken, moet daarvoor de bijbehorende data-sheet raadplegen. Tijdens het initialiseren van het systeem moet de programmeur het controle register van de PIO laden met de voor zijn toepassing juiste waarde.

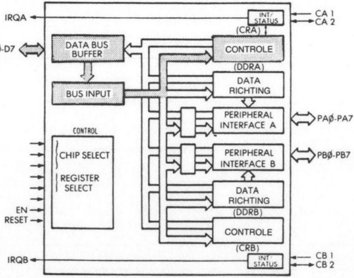


Fig. 7.2: Het gebruik van een PIO. Controle register laden

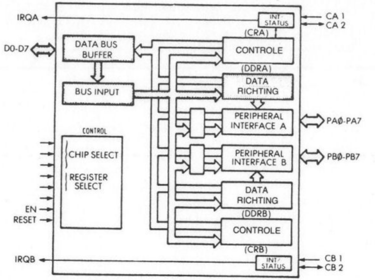


Fig. 7.3: Het gebruik van een PIO. Data richting laden.

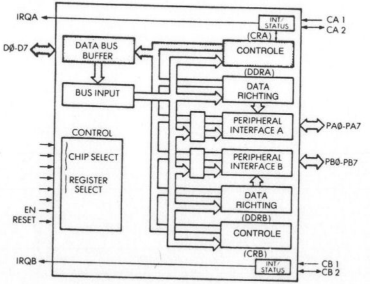


Fig. 7.4: Het gebruik van een PIO. Status lezen.

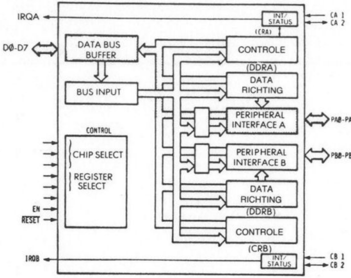


Fig. 7.5: Het gebruik van een PIO. INPUT lezen.

Het programmeren van een PIO

Een typische volgorde voor het programmeren van een PIO is de volgende (de PIO wordt als input gebruikt):

Laad het controle register

Dit is een geprogrammeerde data overdracht van de Z80 (meestal de accumulator) naar het PIO controle register. De opties worden gekozen. Zie figuur 7.2. Dit wordt meestal eenmaal gedaan aan het begin van het programma.

Laad het data-richting register

De richting waarin de I/O lijnen gebruikt worden wordt gespecificeerd. Zie figuur 7.3.

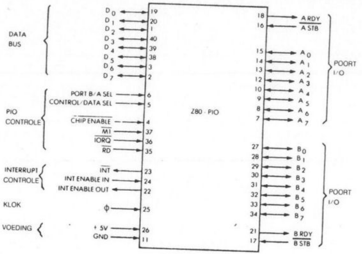


Fig. 7.6: Z80 PIO

Lees de status

Het status register geeft aan, of een geldig byte op de input aanwezig is. Zie figuur 7.4.

Lees de poort

De byte wordt naar de Z80 gestuurd. Zie figuur 7.5.

DE ZILOG Z80 PIO

De Z80 PIO heeft twee poorten, en heeft een architectuur die grotendeels overeenkomt met de zojuist beschreven standaard PIO. Figuur 7.6 geeft aan welke signalen op welke pennen te vinden zijn. Het blokdiagram staat in figuur 7.7.

Iedere PIO poort heeft zes registers: een 8-bits input register, een 8-bits output register, een 2-bits mode-controle register, een 8-bits masker register, een 8-bits data-richting register en een 2-bits masker-controle register. Deze laatste drie registers worden alleen gebruikt, als de poort in bit mode is geprogrammeerd.

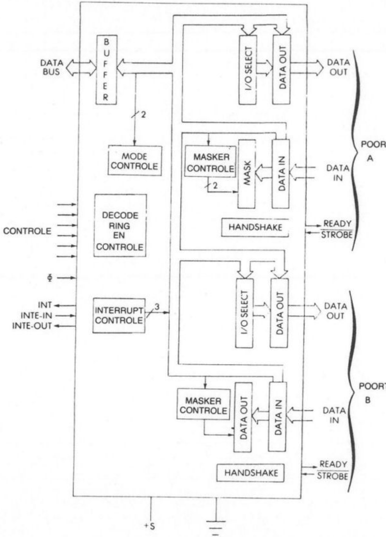


Fig. 7.7: Z80 PIO blokdiagram

De PIO kent vier modes, aangegeven met de inhoud van het mode-controle register (2 bits). Deze zijn: byte output, byte input, byte twee-richtings bus, en bit mode.

De twee bits van de masker-controle registers worden door de programmeur geladen, en geven aan onder welke voorwaarden een interrupt gegenereerd wordt.

Het 8-bits data-richting register geeft van ieder bit de data richting aan, als in die mode gewerkt wordt.

Het programmeren van de Zilog PIO

Een typische volgorde voor het gebruik van de PIO in, bijvoorbeeld, bit mode is:

Laad het mode controle register voor bit mode.

Laad het data-richting register, en specificeer de bits 0-5 als input, en bits 6 en 7 als outputs. De bits zijn van poort A.

Lees een woord uit de input buffer.

Gebruik het maskeer register om de status condities te specificeren.

DE Z80 SIO

De SIO (Seriele Input/Output) is een twee-kanaals chip om asynchrone seriele communicatie mogelijk te maken. De chip bevat een UART (universal asynchronous receiver-transmitter = universele asynchrone ontvanger- zender). De belangrijkste functie van de UART is de serie-parallel en parallel-serie omzetting. Deze chip heeft echter nog enkele andere geavanceerde mogelijkheden. Zoals het automatisch afhandelen van complexe byte georienteerde protocollen (IBM bisync) en twee-bytes georienteerde protocollen (HDLC en SDLC).

Verder kan de chip synchroon werken als een USRT, en CRC codes opwekken en controleren. Een volledige beschrijving valt buiten het bestek van dit boek.

ANDERE I/O CHIPS

Aangezien de Z80 algemeen gebruikt wordt als vervanger van de 8080 kunnen de meeste I/O chips voor de 8080 worden toegepast. Ook andere I/O chips van Zilog zijn toe te passen.

SAMENVATTING

Om efficient van de nu besproken chips gebruikt te kunnen maken, moet de functie van ieder bit, of van iedere groep bits in de verschillende controle registers bekend en begrepen zijn. De ingewikkelde chips nemen de processor enkele taken uit handen, die vroeger d.m.v. software of speciale logica werden uitgevoerd. Dat geldt voor handshake procedures in bijvoorbeeld de SIO. Ook zijn vaak het detecteren en verwerken van interrupts naar deze chips gedelegeerd. Met de informatie gegeven in dit hoofdstuk moet de lezer in staat zijn de basis signalen en registers te begrijpen. Natuurlijk zal het niet bij deze chips blijven. Steeds zullen nieuwe chips, met nog meer ingewikkelde algoritmes, geintroduceerd worden.

8
TOEPASSINGEN (VOORBEELDEN)

INLEIDING

Dit hoofdstuk is bedoeld om de nieuw verworven programmeer bekwaamheden te testen d.m.v. een verzameling voorbeelden van toepassingen. Je zult deze programma's vaak in toepassingen tegenkomen, zo algemeen zijn ze geworden.

We gaan karakters van een randapparaat halen, en zullen deze op verschillende manieren verwerken. Maar voordat we kunnen beginnen maken we eerst een deel van het geheugen schoon (hoewel dat misschien niet nodig is).

HET SCHOONMAKEN VAN EEN DEEL VAN HET GEHEUGEN

We willen een deel van het geheugen schoonmaken (0 maken) van adres BASE + 1 tot BASE + LENGTH, met LENGTH kleiner dan 256.

Het programma:

```
ZEROM  LD    B, LENGTH     LAAD B MET LENGTE
       LD    A,0           MAAK A NUL
       LD    HL, BASE+1    WIJS NAAR BASIS
CLEAR  LD    (HL), A       MAAK ADRES (HL) NUL
       INC   HL            WIJS NAAR VOLGEND ADRES
       DEC   B             TELLER
       JR    NZ, CLEAR     GEDAAN?
       RET
```

In het bovenstaande programma is de lengte van het schoon te maken deel van het geheugen LENGTH. Het register paar HL wijst naar het woord dat 0 moet worden. Register B wordt, zoals gebruikelijk, als teller gebruikt.

De accumulator wordt slechts een maal geladen met de waarde 0, en dan gecopieerd in de opeenvolgende geheugen adressen.

In een geheugen test programma kan met deze routine de inhoud van een bepaalde sectie van het geheugen nul worden gemaakt. Daarna moet het programma testen of alle adressen ook daadwerkelijk nul zijn.

Het bovenstaande programma is recht-toe-recht-aan. Er zijn natuurlijk verbeteringen denkbaar:

```
ZEROM   LD      B,LENGTH
        LD      HL,BASE
LOOP    LD      (HL),0
        INC     HL
        DJNZ    LOOP
        RET
```

Er zijn twee verbeteringen aangebracht: LD A,0 is geelimineerd en er wordt direct de waarde 0 geladen op het adres waar HL naar wijst. En de tweede verbetering is de toepassing vande Z80 instructie DJNZ.

Dit voorbeeld laat zien, *dat een geschreven programma, hoewel het misschien juist is, gewoonlijk verbeterd kan worden*. Ben je eenmaal bekend met de volledige instructie set, dan is het belangrijk dat je dergelijke verbeteringen aanbrengt. Het staat misschien niet alleen beter, maar het programma wordt m.b.t. snelheid, leesbaarheid en grootte ook daadwerkelijk beter.

Oefening 8.1: Schrijf een geheugen test programma dat eerst een blok van 256 woorden 0 maakt, en dan controleert dat het inderdaad nul is geworden. Daarna herhaalt het programma de procedure, maar maakt dan alle bits 1. Dan wordt op ieder adres 01010101 geschreven en gecontroleerd, en tenslotte wordt het zelfde gedaan met 10101010.

Oefening 8.2: Verander het bovenstaande programma zodanig, dat het de geheugen sectie volschrijft met afwisselende nullen en enen. (Eerste adres allemaal nullen, volgende adres allemaal enen, enz.)

POLLING VAN I/O APPARATEN

Stel ons systeem is verbonden met een aantal randapparaten, en d.m.v. polling willen we uitzoeken welk apparaat om attentie vraagt. Hun status registers staan op de adressen IOSTATUS1, IOSTATUS2 en IOSTATUS3. Het programma gaat als volgt:

```
TEST  IN    A, (STATUS1)        LEES STATUS1
      BIT   7, A                TEST "READY" BIT (BIT 7)
      JP    NZ, FOUND1          SPRING NAAR ROUTINE1
      IN    A, (STATUS2)        HET ZELFDE VOOR
            IN A, (STATUS3)     APPARAAT 2
      BIT   7,A
      JP    NZ,FOUND2
      IN    A, (STATUS3)        EN VOOR APPARAAT 3
      BIT   7,A
      JP    NZ,FOUND3
      (foutbehandeling)
```

Het masker bevat "10000000" als we positie 7 testen. Het resultaat van de BIT instructie is, dat het Z bit van het status register 1 geset wordt als "MASK AND STATUS" niet nul is, d.w.z. als het corresponderende bit van STATUS gelijk is aan die in MASK (het masker). De JP NZ instructie (spring als ongelijk aan nul) resulteert in een sprong naar de juiste FOUND routine.

HET BINNENHALEN VAN KARAKTERS

Stel dat we gevonden hebben, dat er een karakter klaar staat bij het toetsenbord. Deze karakters worden in een gedeelte van het geheu-

```
STRING LD    HL, BUFFER   WIJS NAAR BUFFER
NEXT   CALL  GETCHAR      HAAL KARAKTER
       CP    SPC          IS HET SPC?
       JR    Z, OUT       JA? KLAAR
       LD    (HL), A      ZET KARAKTER IN BUFFER
       INC   HL           VOLGEND BUFFER ADRES
       JR    NEXT         HAAL VOLGEND KARAKTER
OUT    RET                KLAAR
```

gen, BUFFER, opgespaard totdat een speciaal karakter SPC verschijnt. We nemen aan dat de code van dat karakter eerder al gedefinieerd is.

De subroutine GETCHAR haalt 1 karakter van het toetsenbord en laadt het in de accumulator. (Zie hoofdstuk 6 voor meer details) We nemen aan dat maximaal 256 karakters op deze wijze in de accumulator geladen worden voordat we het karakter SPC tegenkomen.

Oefening 8.3: Deze basis routine gaan we verbeteren:
a - Echo het karakter naar het randapparaat (bijvoorbeeld een Teletype).
b - Controleer dat het aantal karakters niet groter dan 256 is.

De karakters staan nu in de geheugen buffer. Ze kunnen op meerdere manieren verwerkt worden.

HET TESTEN VAN EEN KARAKTER

We willen testen of een karakter op adres LOC gelijk is aan 0, 1 of 2:

```
ZOT    LD    A, (CHAR)      HAAL KARAKTER
       CP    00             IS HET NUL?
       JP    Z, ZERO        JA? GA NAAR ZERO
       CP    01             IS HET 1?
       JP    Z,ONE
       CP    02             IS HET 2?
       JP    Z,TWO
       JP    NOT FND        NIET GEVONDEN
```

Het karakter wordt eenvoudig gelezen, en met de CP instructie wordt zijn waarde getest.

We gaan door met een andere test.

CATEGORIE TEST

We onderzoeken of het ASCII karakter op adres LOC een getal tussen 0 en 9 is:

```
BRACK  LD    A,(CHAR)    HAAL KARAKTER
       AND   7FH         MASKEER PARITEITS BIT
       CP    30H         ASCII 0
       JR    C, OUT      KARAKTER TE LAAG?
       CP    39H         ASCII 9
       JR    NC, OUT     KARAKTER TE HOOG?
       CP    A           FORCEER HET Z BIT
OUT    RET               GEDAAN
```

ASCII "0" wordt hexadecimaal voorgesteld door 30 of D0, afhankelijk van het feit of het pariteits bit al of niet wordt gebruikt. ASCII "9" is 39 of B9.

De tweede instructie haalt bit 7, het pariteits bit, weg, zodat het programma geldig is voor beide voorstellingswijzen. De waarde van het karakter wordt dan vergeleken met de ASCII karakters 0 en 9. De Z vlag wordt geset (1 gemaakt) als de vergelijking slaagt. Het carry bit wordt geset in geval een borrow optreedt, en anders gereset. M.a.w. het carry bit wordt geset als de waarde van het getal in de instructie groter is dan dat in de accumulator. Het wordt gereset (0) als het kleiner of gelijk is.

De laatste instructie maakt de Z vlag gelijk aan 1, om het aanroepende programma aan te duiden, dat het karakter inderdaad tussen 0 en 9 lag. Andere afspraken kunnen ook gehanteerd worden, zoals het laden van een getal in de accumulator.

Oefening 8.4: Is het volgende programma gelijk aan het bovenstaande?

```
LD    A, (CHAR)
SUB   30H
JP    M, OUT
SUB   10
JP    P, OUT
ADD   10
```

Oefening 8.5: Onderzoek of een karakter in de accumulator een letter van het alfabet is.

In ASCII tabellen wordt vaak de pariteit gebruikt. Bijvoorbeeld: de ASCII code voor "0" is 0110000, een 7-bits code. Gebruiken we echter oneven pariteit, het aantal enen in een woord moet oneven zijn, dan wordt de code 10110000. Een extra 1 wordt links toegevoegd. De hexadecimale code wordt: B0. We hebben een programma nodig dat de pariteit berekent.

PARITEITS BEREKENING

Dit programma berekent de even pariteit en plaatst deze op bit positie 7:

```
PARITY  LD    A, (CHAR)    HAAL KARAKTER
        AND   7FH          MAAR PARITEITS BIT 0
        JR    PE, OUT      KLAAR,ALS PARITEIT
                           EVEN IS
        OR    80H          ZET RESULTAAT WEG
OUT     LD    (LOC), A     STORE RESULT
```

Dit programma maakt gebruik van het interne pariteits schakeling in de Z80.

De derde instructie JR PE,OUT test of het woord in de accumulator al even pariteit heeft. Is dat het geval, dan kan naar het einde OUT gesprongen worden.

Is de pariteit niet even, er is dan niet gesprongen, dan is de pariteit oneven, en moet een 1 in positie 7 worden geplaatst. Dat doet de vierde instructie:

```
OR 80H
```

Tenslotte wordt het getal met de juiste pariteit op adres LOC gezet.

Oefening 8.6: Het bovenstaande probleem was eigenlijk te eenvoudig om op te lossen. Los nu het zelfde probleem op, zonder gebruik te maken van het pariteits circuit van de Z80. Schuif de inhoud van de accumulator en tel het aantal enen. Plaats daarna de juiste waarde in positie 7.

Oefening 8.7: Controleer of een woord de juiste pariteit heeft. Maak daarbij gebruik van het bovenstaande programma. Bereken de pariteit, en vergelijk deze met de verwachte pariteit.

CODE OMZETTING: ASCII NAAR BCD

Het omzetten van ASCII naar BCD is erg eenvoudig. De hexadecimale code van de ASCII karakters 0 tot 9 is 30 tot 39, of B0 tot B9, afhankelijk van de pariteit. De BCD voorstelling wordt dus verkregen door de 3 of de B weg te halen, d.w.z. maskeer de linker vier bits.

```
ASCBCD CALL  BRACK           CONTROLEER OF
                             KARAKTER 0-9 IS
       JP    NZ, ILLEGAL     ILLIGAAL KARAKTER
       LD    A, (CHAR)       HAAL KARAKTER
       AND   0F              MASKEER LINKER 4 BITS
       LD    (BCDCHAR), A    BERG RESULTAAT OP
```

Oefening 8.8: Schrijf een programma dat BCD in ASCII omzet.

Oefening 8.9: Schrijf een programma dat BCD omzet in binair (dit is moeilijker)
Hint: $N_3N_2N_1N_0$ in BCD is $(((N_3 \times 10) + N_2) \times 10 + N_1) \times 10 + N_0$ binair.
Doe voor de vermenigvuldiging met 10 als volgt: schuif naar links (= ×2), schuif naar links (= ×4), een ADC (= ×5), schuif naar links (= ×10).

In de volledige BCD notatie bevat het eerste woord het aantal BCD cijfers, de volgende nibble bevat het teken, en iedere volgende nibble bevat een BCD cijfer. We nemen aan dat er geen decimale komma in voorkomt. De laatste nibble van een blok kan ongebruikt zijn.

OMZETTING VAN HEX NAAR ASCII

"A" bevat een hexadecimaal cijfer. We moeten eenvoudig een 3 of een B bij de linker nibble optellen:

```
AND   FH        MAAK LINKER NIBBLE NUL
ADD   A, 30H    ASCII
CP    A, 3AH    CORRECTIE NODIG?
JP    M,OUT
ADD   A, 7      CORRECTIE VOOR A TOT F
```

Oefening 8.10: Zet HEX om in ASCII. A bevat twee HEX getallen

HET GROOTSTE GETAL IN EEN TABEL

Het begin adres van de tabel is te vinden op adres BASE op pagina 0. Het eerste woord in de tabel geeft het aantal bytes in die tabel. Dit programma zoekt het grootste element van de tabel. Deze waarde komt in A, en zijn positie wordt op adres INDEX geladen.

Dit programma gebruikt de registers A, B, H en L. Bovendien maakt het gebruik van indirecte adressering, zodat het een tabel overal in het geheugen kan onderzoeken (Zie figuur 8.1).

```
MAX        LD   HL, BASE       TABEL ADRES
           LD   B, (HL)        AANTAL BYTES IN TABEL
           LD   A, 0           MAAK MAXIMUM
                               WAARDE 0
           LD   (INDEX), HL    INITIALISEER INDEX
           INC  HL             VOLGEND WOORD
LOOP       CP   (HL)           VERGELIJK WOORD
           JR   NC, NOSWITCH   SPRING ALS KLEINER
                               DAN MAX
           LD   A, (HL)        LAAD NIEUWE MAX
                               WAARDE
           LD   (INDEX), HL    LAAD NIEUWE ADRES
                               VAN MAX
NOSWITCH   INC  HL             WIJS NAAR VOLGEND
                               WOORD
           DEC  B              TELLER
           JR   NZ, LOOP       GA DOOR TOT B = 0
           RET
```

Dit programma test eerst het n-de woord. Is het groter dan 0, dan komt het woord in A, en het adres wordt in INDEX geladen. Daarna wordt het (n−1)-de woord getest, enz. Het programma werkt alleen voor gehele positieve getallen.

Oefening 8.11: Verander het programma zodanig, dat het ook werkt voor negatieve getallen in twee-complement.

Oefening 8.12: Werkt het programma ook voor ASCII karakters?

Oefening 8.13: Schrijf een programma, dat n getallen sorteert volgens grootte.

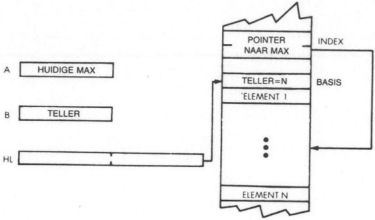


Fig. 8.1: Het grootste element van een tabel

Oefening 8.14: Schrijf een programma, dat n namen van elk drie karakters sorteert op alfabetische volgorde.

DE SOM VAN N ELEMENTEN

Dit programma berekent de 16-bits som van N woorden van een tabel. Het start adres van de tabel staat op adres BASE, op pagina 0. Het eerste woord is het aantal op te tellen woorden N. Het 16-bits resultaat komt te staan op de geheugen adressen SUMLO en SUMHI. Is de som groter dan 16 bits, dan worden alleen de laagste orde 16 bits bewaard. (De hoogste orde bits worden afgekapt). Het programma verandert de registers A, B, H, L en IX. Het maximale aantal elementen is 256. Zie figuur 8.2.

```
SUMIG   LD   HL, BASE     WIJS NAAR BASIS TABEL
        LD   B, (HL)      LENGTE IN TELLER B
        INC  HL           WIJS NAAR EERSTE WOORD
        LD   IX, SUMLO    WIJS NAAR RESULTAAT,
                          LAAG
        LD   A, 0         MAAK RESULTAAT 0
```

```
         LD   (IX+0), A        LAAG
         LD   (IX+1), A        EN HOOG
ADLOOP   LD   A, (HL)          HAAL WOORD
         ADD  A, (IX+0)        BEREKEN PARTIELE SOM
         LD   (IX+0), A        BERG HET OP
         JR   NC, NOCARRY      TEST CARRYBIT
         INC  (IX+1)           TEL CARRY OP BIJ HOGE
                               BYTE
NOCARRY  INC  HL               WIJS NAAR VOLGENDE
                               WOORD
         DEC  B                TELLER MIN 1
         JR   NZ, ADLOOP       TEL OP TOT HET EINDE
         RET
```

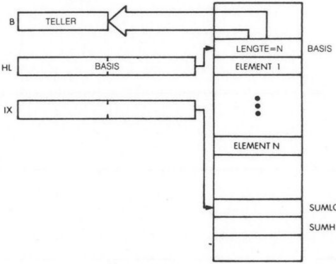


Fig. 8.2: Som van N elementen

Het programma is recht-toe-recht-aan, en is gemakkelijk te begrijpen.

Oefening 8.15: Verander het programma zodanig, dat het
a - een 24-bits som berekent,
b - een 32-bits som berekent, en
c - een overflow detecteert.

EEN CONTROLE SOM BEREKENING

Een controle som is een cijfer of een getal, dat berekend is uit een blok opeenvolgende karakters. Deze som wordt berekend op het moment dat de data weggeborgen wordt, en wordt eveneens mee opgeborgen aan het eind. Bij het weer ophalen van de data wordt deze controle som opnieuw berekend, en vergeleken met de opgeborgen controle som. Een verschil in deze twee waarden betekent een fout.

Er zijn vele algoritmen in gebruik. Wij zullen een exclusieve-OF gebruiken van alle elementen van de tabel, en het resultaat in de accumulator laden. De basis van de tabel staat op adres BASE op pagina nul. Het eerste woord in de tabel geeft het aantal woorden in die tabel. Het programma verandert A, B, H, en L. N moet kleiner dan 256 zijn.

```
CHECKSUM  LD   HL, BASE        BASISADRES TABEL
          LD   B, (HL)         HAAL N
          XOR  A               CONTROLE SOM WORDT 0
          INC  HL              WIJS NAAR EERSTE
                               ELEMENT
CHLOOP    XOR  (HL)            BEREKEN CONTROLE SOM
          INC  HL              WIJS NAAR VOLGENDE
                               ELEMENT
          DEC  B               TELLER 1 KLEINER
          JR   NZ, CHLOOP      GA DOOR TOT EINDE
          LD   (CHCKSM), A     BEWAAR CONTROLE SOM
          RET
```

TEL DE NULLEN

Dit programma telt het aantal nullen in onze tabel, en plaatst dit aantal op adres TOTAL. Het programma verandert A, B, C, H, en L.

```
ZEROS  LD   HL, BASE      WIJS NAAR TABEL
       LD   B, (HL)       LENGTE VAN TABEL IN TELLER
       LD   C, 0          TOTAAL WORDT 0
       INC  HL            WIJS NAAR EERSTE WOORD
ZLOOP  LD   A, (HL)       HAAL ELEMENT
       OR   0             SET ZERO VLAG
       JR   NZ, NOTZ      IS HET EEN 0?
       INC  C             JA?TEL 1 OP BIJ C
NOTZ   INC  HL            WIJS NAAR VOLGENDE WOORD
       DEC  B             TREK 1 AF VAN TELLER
       JR   NZ,ZLOOP
       LD   A,C           REDT TOTAAL
       LD   (TOTAAL), A   BERG TOTAAL OP
```

Oefening 8.16: Verander het programma, zodat het
a - het aantal sterren (*) telt,
b - het aantal letters van het alfabet telt, en
c - het aantal getallen tussen 0 en 9 telt.

BLOK VERPLAATSING

We gaan ieder derde byte uit het blok op adres FROM verplaatsen naar een blok, dat op adres TO begint.

```
FER3   LD    HL,FROM
       LD    DE, TO      INITIALISEER POINTERS
       LD    BC,SIZE
LOOP   LDI               AUTOMATISCHE VERPLAATSING
       INC   HL
       INC   HL          SLA TWÈE BYTESOVER
       JR    PE,LOOP
```

BCD BLOK VERPLAATSING

We willen BCD cijfers in het geheugen "pushen", d.w.z. 4-bits nibbles moeten verschoven worden (Zie figuur 8.3). Hier volgt het programma:

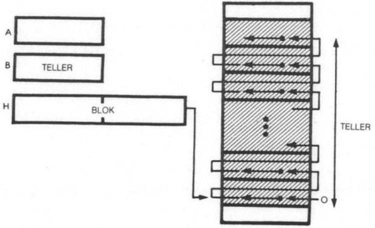


Fig. 8.3: BCD blok verplaatsing. Het geheugen

```
DMOV  LD    B,COUNT
      LD    HL,BLOCK
      XOR   A          A WORDT 0
LOOP  RLD
      INC   HL         WIJS NAAR VOLGENDE BYTE
      DJNZ  LOOP       VERLAAG TELLER TOT DEZE 0 IS
```

Het programma gebruikt de RLD instructie, die wij nog niet kennen. RLD roteert een BCD cijfer links tussen A en (HL). (HL) of M is de inhoud van het geheugenadres waar HL naar wijst.

M,laag gaat naar M,hoog
M,hoog gaat naar A,laag
A,laag gaat naar M,laag.

Hoog en laag hebben hier betrekking op 4-bits nibbles.

Om de krachtige DJNZ instructie te kunnen gebruiken, is register B cijfer teller. HL wijst naar het begin van het blok. Het linker cijfer, dat uitgeschoven wordt door de rotatie tussen twee opeenvolgende toegangen tot het blok, wordt in A bewaard.

Onder in het blok worden nullen toegevoegd.

DE VERGELIJKING VAN TWEE 16-BITS GETALLEN MET TEKEN

IX wijst naar het eerste getal N1.

IY wijst naar N2 (zie figuur 8.4).

Het programma set het carry bit als N1 < N2, en set de Z vlag als N1 = N2.

```
COMP    LD    B, (IX+1)    HAAL TEKEN VAN N1
        LD    A, B
        AND   80H          TEST TEKEN, MAAK CY 0
        JR    NZ, NEGM1    N1 IS NEGATIEF
        BIT   7, (IY+1)
        RET   NZ           N2 IS NEGATIEF
        LD    A, B
        CP    (IY+1)       TEKENS ALLEBEI POSITIEF
        RET   NZ
        LD    A, (IX)
        CP    (IY)
        RET
NEGM1   XOR   (IY + 1)
        RLA                TEKENBIT IN CY
        RET   C            TEKENS ONGELIJK
        LD    A, B
        CP    (IY+1)       BEIDE TEKENS NEGATIEF
        RET   NZ
        LD    A, (IX)
        CP    (IY)
        RET
```

Dit programma test de tekens van N1 en N2. Als N1 negatief is, wordt naar NEGM1 gesprongen. Anders wordt het bovenste gedeelte van het programma uitgevoerd.

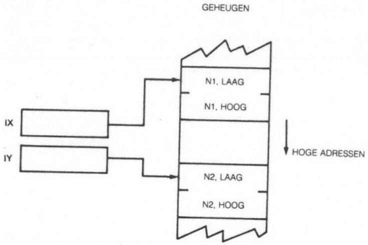


Fig. 8.4: Vergelijking van twee getallen

In de vijfde regel wordt het tekenbit van N2 direct in het geheugen getest d.m.v. de instructie

```
        BIT    7, (IY + 1)
```

Het zelfde zouden we kunnen doen bij N1, ware het niet, dat we dit getal nog nodig hebben. Daarom is het gemakkelijker N1 in B te laden:

```
COMP    LD     B, (IX + 1)
```

Het getal moet in B worden bewaard, omdat de AND instructie de inhoud van A kan vernietigen.

```
        LD     A, B
        AND    80H
```

Merk op dat een RET in regel 6 is toegevoegd:

```
        RET    NZ
```

Deze eigenschap van de Z80 maakt het programmeren eenvoudiger.
De vergelijk instructie vergelijkt direct de inhoud van het geheugen:

```
CP      (IY + 1)
```

De meest significante bytes worden eerst vergeleken, daarna de minst significante.

Het uitgebreide gebruik van geindexeerde adressering resulteert in een efficiente code.

BUBBLE-SORT

Bubble-sort is een sorteer techniek, die gebruikt wordt om de elementen van een tabel in stijgende of dalende volgorde te rangschikken. De bubble-sort techniek (bubble = luchtbel) ontleent zijn naam aan het feit, dat het kleinste element als een luchtbel naar de "oppervlakte" van de tabel opstijgt. Iedere keer als het botst met een groter element, springt het er overheen en vervolgt zijn weg naar boven.

Een voorbeeld van bubble-sort wordt in figuur 8.5 getoond. De te sorteren lijst bevat de elementen: (10, 5, 0, 2, en 100). De lijst moet in dalende volgorde gesorteerd worden, d.w.z. het kleinste element boven. Het algoritme is eenvoudig, en het stroomdiagram is in figuur 8.7 te vinden.

De twee bovenste (of anders de twee onderste) elementen worden vergeleken. Is het onderste element kleiner dan het bovenste, dan worden ze verwisseld. Anders niet. Voor praktische doeleinden wordt m.b.v de vlag "EXCHANGED" onthouden, of er een verwisseling heeft plaats gevonden. Dit proces wordt herhaald met het volgende paar elementen, enz., totdat alle elementen twee aan twee vergeleken zijn.

Deze eerste stap wordt geillustreerd door de plaatjes 1, 2, 3, 4, 5, en 6 van figuur 8.5.

Als geen enkel element wordt verwisseld, is het sorteren voltooid. Is er wel een element verwisseld, dan wordt alles nog eens herhaald.

In figuur 8.6 is te zien, dat er vier stappen nodig zijn in dit voorbeeld.

Het proces is eenvoudig, en wordt veel gebruikt.

Er doet zich een probleempje voor bij het verwisselen van de elementen. We kunnen bij dit verwisselen niet zeggen:

A = B
B = A

want de oorspronkelijke waarde van A gaat daarmee verloren. (Probeer het maar eens met een voorbeeld.)

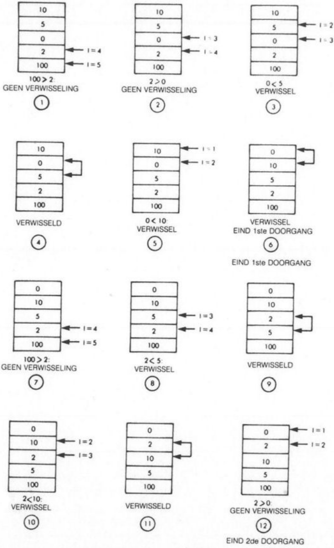


Fig. 8.5: Bubble-sort voorbeeld: Fase 1 tot 12

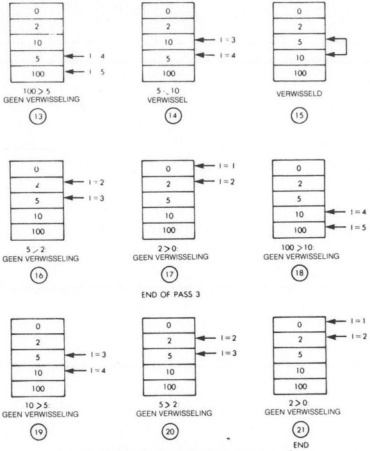


Fig. 8.6: Bubble-sort voorbeeld: Fase 13 tot 21

De juiste oplossing is het gebruik van een tijdelijke variabele of een tijdelijk adres om de waarde van A te bewaren:

```
TEMP = A
A    = B
B    = TEMP
```

Probeer dit ook, en je zult zien dat het werkt. Dit is de manier om waarden tussen variabelen te verwisselen en wordt in alle programma's gebruikt. Zie figuur 8.7.

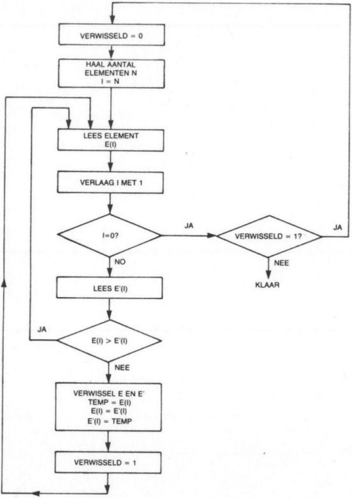


Fig. 8.7: Bubble-sort stroomdiagram

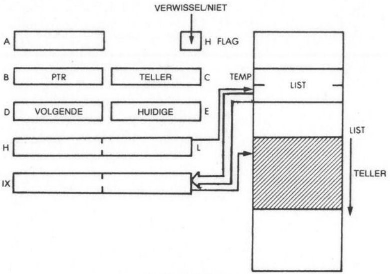


Fig. 8.8: Bubble-sort

Figuur 8.8 geeft aan welke registers en geheugen adressen gebruikt worden.

```
BUBBLE   LD   (TEMP), HL     TEMP=(HL)
AGAIN    LD   IX, (TEMP)     (IX)=(HL)
         RES  FLAG, H        VLAG "EXCHANGED"=0
         LD   B, C
         DEC  B
NEXT     LD   A, (IX)
         LD   D, A           D = HUIDIGE ELEMENT
         LD   E, (IX+1)      E = VOLGEND ELEMENT
         SUB  E              VERGELIJK
         JR   NC, NOSWIT     GA NAAR NOSWIT
                             ALS ELEMENT
                             GROTER IS DAN
                             VOLGEND ELEMENT
XCHANG   LD   (IX), E        VERWISSEL HUIDIG EN
         LD   (IX+1), D      VOLGEND EL.
         SET  FLAG, H        EXCHANGED VLAG = 1
```

```
NOSWIT  INC   IX          VOLGENDE ELEMENT
        DJNZ  NEXT        GA DOOR TOT B = 0
        BIT   FLAG, H     IS ER VERWISSELD?
        JR    NZ, AGAIN   JA? HERHAAL DAN
        RET
```

SAMENVATTING

In dit hoofdstuk zijn een aantal algemeen bruikbare routines getoond, waarin de tot nu toe besproken technieken gebruikt zijn. Met deze kennis ben je gereed om met het programmeren te beginnen. Veel van deze routines gebruiken een speciale data structuur, de tabel. Andere soorten data structuren zullen in het volgende hoofdstuk worden besproken.

9
DATA STRUCTUREN

DEEL 1 – THEORIE

INLEIDING

Het maken van een goed programma omvat twee taken: het ontwerpen van een *algoritme,* en het ontwerpen van de *data structuren.* In de meeste eenvoudige programma's komen geen belangrijke data structuren voor, zodat het voornaamste doel van het leren programmeren is: het ontwerpen van de algoritmen en het efficient coderen van deze algoritmen in een gegeven machine taal. Dat is wat we tot nu toe ook gedaan hebben. Echter, in de meer ingewikkelde programma's komen data structuren voor, en die moeten we begrijpen. We hebben al kennis gemaakt met twee data structuren: de tabel en de stapel. In dit hoofdstuk komen andere, meer algemene, data structuren aan de orde. Wat nu volgt is theorie, en volkomen onafhankelijk van welke microprocessor dan ook. Deze theorie betreft de logische organisatie van de data in een systeem. Er zijn specialistische boeken geschreven over data structuren, net als er boeken geschreven zijn over vermenigvuldigen, delen en andere algoritmen. In dit hoofdstuk zullen we ons daarom tot de hoofdzaken beperken. Het pretendeert ook niet compleet te zijn. We gaan nu kijken naar de meest algemene data structuren.

POINTERS

Een pointer is een getal dat gebruikt wordt om het adres van bepaalde data aan te wijzen. Iedere pointer is een adres. Ieder adres is echter nog geen pointer. Een adres is alleen dan een pointer als deze naar data

of naar gestructureerde informatie wijst. We hebben al een typische pointer gezien: de stack pointer. Later zullen we zien, dat een stapel een algemene data structuur is: de LIFO.

Een ander voorbeeld van een pointer is een indirect adres bij indirecte adressering. Dat adres wijst naar de data, die bewerkt moet worden.

Oefening 9.1: Bekijk figuur 9.1. Op adres 15 staat een pointer naar tabel T. Tabel T start op adres 500. Wat is de waarde van deze pointer naar T?

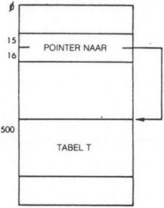


Fig. 9.1: Indirecte pointer

LIJSTEN

Bijna elke data structuur is georganiseerd in een of andere vorm van een lijst.

Sequentiele lijsten

Een sequentiele lijst, of tabel, of blok, is waarschijnlijk de meest eenvoudige data structuur, en is de data structuur die we al hebben gebruikt. Tabellen worden gewoonlijk geordend volgens een bepaald criterium, zoals het alfabet of toenemende grootte. Daardoor is het ge-

makkelijk een element uit de tabel te halen, gebruik makende van, bijvoorbeeld, geindexeerde adressering. Een blok heeft meestal betrekking op een groep, niet geordende, data met vaste grenzen. Het kan een reeks karakters bevatten, of het is een sector op een disk, of het is een gebied (het heet dan een segment) van het geheugen. In dergelijke gevallen is het soms niet gemakkelijk een bepaald element midden in het blok te vinden.

Om dat toch mogelijk te maken worden aanwijzers, "directories" genaamd, gebruikt.

Directories

Een directory is een lijst van tabellen of blokken. Bijvoorbeeld, in file systemen wordt normaal gebruik gemaakt van directory structuren. (Een file systeem wordt gebruikt om allerlei gegevens op te slaan.) Allereerst is er een hoofd directory met daarin de gebruikers van het file systeem. Zie figuur 9.2. Stel we willen iets weten over gebruiker "John". In de hoofd directory zoeken we het vakje John op. Dat vakje bevat een wijzer naar de directory van John, met daarin de wijzers naar de gegevens van John. Er wordt in dit systeem dus gebruik gemaakt van een twee-niveau directory systeem. Een flexibel directory systeem laat vaak het tussenvoegen van andere directories toe, als dat het gemak van de gebruiker dient.

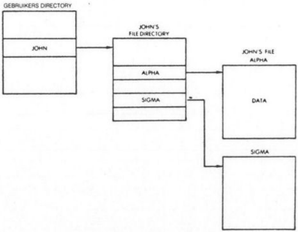


Fig. 9.2: Een directory structuur

Geketende lijsten

In een systeem komen vaak blokken informatie die data, gebeurtenissen of andere structuren voorstellen, voor, die niet gemakkelijk te verplaatsen zijn. Indien dat mogelijk zou zijn, zouden we ze waarschijnlijk rangschikken in een tabel om ze te sorteren. Het probleem is nu, hoe we de elementen op hun plaats kunnen houden en ze toch ordenen. De oplossing voor dit probleem is de geketende lijst. Het principe ervan wordt in figuur 9.3 geillustreerd. We zien dat een pointer EERSTEBLOK, naar het begin van het blok wijst. Een bepaald adres in blok 1 bevat een pointer naar blok 2. Dit proces gaat zo door voor blok 2 en blok 3. Omdat blok 3 het laatste blok is, bevat zijn pointer een speciale "nil" waarde, of wijst naar zichzelf. Dit is een economische structuur, want er zijn slechts enkele pointers nodig (1 per blok), en de gebruiker hoeft geen blokken in het geheugen te verplaatsen.

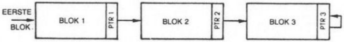


Fig. 9.3: Een geketende lijst

Laten we eens gaan kijken naar de manier waarop een nieuw blok wordt toegevoegd. Zie figuur 9.4. Stel het nieuwe blok staat op adres NIEUWBLOK, en moet tussen blok 1 en blok 2 komen. De pointer PTR1 krijgt gewoon de waarde NIEUWBLOK. Deze wijst dus nu naar blok X. PTRX krijgt de oude waarde van PTR1 en wijst dus naar blok 2. De andere pointers blijven ongewijzigd. Voor het toevoegen van een nieuw blok hoeven slechts twee wijzers veranderd te worden. Efficient dus.

Oefening 9.2: Teken een diagram dat aangeeft hoe blok 2 weggehaald wordt uit deze structuur.

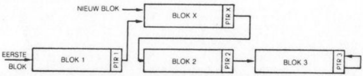


Fig. 9.4: Toevoegen van een nieuw blok

Vele soorten lijsten zijn er ontwikkeld met ieder zijn eigen specifieke eigenschappen. We gaan enkele van de meest gebruikte soorten geketende lijsten nader bekijken.

Queue

Een queue heet officieel een FIFO, of een first-in-first-out lijst. Zie figuur 9.5. Het linker blok is een subroutine voor een randapparaat, bijvoorbeeld een printer. De blokken rechts in de tekening zijn de verzoeken van verschillende programma's of routines om karakters te mogen printen. De volgorde waarin deze verzoeken worden behandeld wordt door de queue bepaald. (Een queue is een rij.) Allereerst is blok 1 aan de beurt, daarna blok 2 en dan blok 3. De afspraak in een queue is, dat ieder nieuw blok in de queue toegevoegd wordt aan het einde van de queue. In ons geval dus na PTR3. Het eerste blok wordt het eerst geholpen: first-in-first-out. Een queue komt algemeen voor in computer systemen, als meerdere gebeurtenissen schaarse onderdelen van het systeem moeten delen, zoals randapparatuur, en de processor zelf.

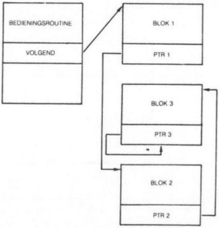


Fig. 9.5: Een queue

Stapel

De stapel hebben we al tot in details bestudeerd. Het is een last-in-first-out structuur (LIFO). Het laatst toegevoegde element wordt weer het eerst van de stapel gehaald. Een stapel kan worden uitgevoerd als een gesorteerd blok of als een lijst. Omdat de meeste microprocessor stapels bedoeld zijn voor snelle gebeurtenissen, zoals subroutines en interrupts, wordt meestal een continu blok gebruikt i.p.v. een geketende lijst.

Geketende lijst contra blok

De queue kan op de zelfde wijze uitgevoerd worden als een blok met gereserveerde adressen. Het voordeel van een continu blok is de korte zoektijd en het ontbreken van pointers. Het nadeel is, dat het blok meestal groot is, want het moet de maximale grootte van de structuur kunnen bevatten. Het is ook moeilijk elementen toe te voegen of uit te wissen bij een blok. Omdat geheugen niet onbeperkt groot kan zijn, worden blokken gereserveerd voor structuren met vaste afmetingen, of voor structuren waarbij snelheid een rol speelt, zoals bij stapels.

Circulaire lijst

Een circulaire lijst is een geketende lijst, waarin het laatste element naar het eerste wijst. Dat wordt in figuur 9.6 geillustreerd. Bij dit soort lijsten wordt vaak een pointer gebruikt, die naar het gebruikte blok wijst. Voor het uitvoeren van de volgende gebeurtenis wordt deze pointer een positie naar links of naar rechts geschoven. In een dergelijke structuur hebben alle blokken meestal de zelfde prioriteit. Circulaire lijsten kunnen ook voorkomen binnen andere structuren, om het opzoeken van het eerste blok na het laatste blok eenvoudiger te maken.

Een polling programma maakt gewoonlijk gebruik van een circulaire lijst.

Bomen

Als er een logisch verband bestaat tussen de elementen van een structuur (gewoonlijk wordt dat syntax genoemd), kan een boom structuur worden gebruikt. Een eenvoudig voorbeeld is de stamboom. Zie figuur 9.7. Smith heeft twee kinderen: een zoon, Robert, en een dochter, Jane. Jane op haar beurt heeft drie kinderen: Liz, Tom en

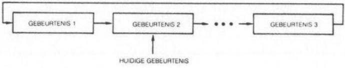


Fig. 9.6: Circulaire lijst

Phil. Tenslotte heeft Tom ook weer twee kinderen: Max en Chris. Robert heeft helemaal geen nakomelingen.

Dit is een gestructureerde boom. In figuur 9.2 zijn we al een eenvoudige boom tegen gekomen. De directory structuur is een twee-niveau boom. Bomen kunnen met voordeel gebruikt worden, daar waar elementen volgens een vaste structuur geklassificeerd kunnen worden. Toevoegen en opzoeken van elementen is goed mogelijk. Informatie die later nog bewerkt moet worden, bijvoorbeeld in vertalers, wordt vaak in een boom structuur opgeslagen.

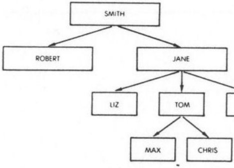


Fig. 9.7: Stamboom

Dubbel-geketende lijsten

Er kunnen meerdere verbindingen tussen de elementen van een lijst gelegd worden. Het eenvoudigste voorbeeld is de dubbel-geketende lijst. Figuur 9.8 laat dit zien. We hebben nu niet alleen pointers van links naar rechts, maar ook van rechts naar links. Toegang tot een vorig of een volgend blok is eenvoudig geworden. Dit kost wel een extra pointer per blok.

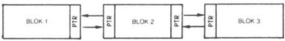


Fig. 9.8: Dubbel-geketende lijst

ZOEKEN EN SORTEREN

Het zoeken en sorteren van elementen van een lijst hangt direct af van het type structuur dat voor de lijst gebruikt is. Voor de meest gebruikte data structuren zijn vele zoek algoritmen ontwikkeld. De geindexeerde adressering hebben we al gebruikt. Dat is mogelijk, als de elementen volgens een bekend criterium gerangschikt zijn. Dergelijke elementen zijn d.m.v. hun nummer op te zoeken.

Bij *lineair zoeken* wordt het hele blok doorzocht. Het is een inefficiente techniek, die soms gebruikt moet worden bij gebrek aan beter, omdat de elementen niet geordend zijn.

Binair of logarithmisch zoeken gebeurt door in een gesorteerde lijst het zoek interval iedere stap door twee te delen. Stel we zoeken in een alfabetische lijst. We beginnen in het midden, en kijken of de gezochte naam voor of na dit punt staat. Is dit na het midden, dan elimineren we de eerste helft van de lijst. De overgebleven helft wordt op de zelfde wijze onderzocht, enz. De maximale lengte van de zoekactie is $\log_2 n$, waarbij n het aantal elementen van de tabel is.

Er bestaan nog vele andere technieken.

SAMENVATTING

Dit deel van het hoofdstuk is uitsluitend bedoeld als een kennismaking met de data structuren die door de programmeur gebruikt kunnen worden. De meeste data structuren hebben een eigen naam gekregen, maar in ingewikkelde systemen kan iedere mogelijke combinatie van deze structuren gebruikt worden. Het aantal mogelijkheden wordt alleen begrenst door de verbeelding van de programmeur. Vele zoek en sorteer technieken zijn ontwikkeld om toegepast te worden op deze data structuren. Een uitgebreide beschrijving daarvan valt echter buiten het bestek van dit boek. Het doel van dit hoofdstuk is het belang van de juiste data structuur voor een bepaalde toepassing aan te tonen.

We zijn nu toe aan gedetailleerde voorbeelden van programma's.

DEEL 2 – ONTWERP VOORBEELDEN

INLEIDING

In dit deel zullen voorbeelden van ontwerpen van de tabel, gesorteerde lijst, en geketende lijst besproken worden. We zullen zoek, tussenvoeg, en uitwis algoritmen programmeren voor deze structuren.

De lezer die in deze geavanceerde technieken geinteresseerd is, moet de programma's zeker tot in detail analiseren.

De beginnende programmeur kan dit deel echter in eerste instantie best overslaan, en pas doornemen als hij denkt, dat hij er klaar voor is.

De principes die besproken zijn in het eerste deel moeten goed begrepen zijn, om de voorbeelden te kunnen volgen. De programma's maken niet alleen gebruik van alle adresserings methoden van de Z80, maar ook van vele principes en technieken uit de vorige hoofdstukken.

We gaan nu drie structuren behandelen: een gewone lijst, een alfabetische lijst, en een geketende lijst met directory. Voor iedere structuur worden drie programma's ontwikkeld: zoeken, tussenvoegen, en wissen.

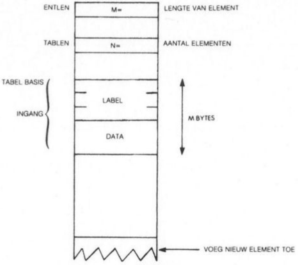


Fig. 9.9: De tabelstructuur

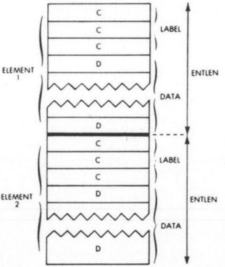
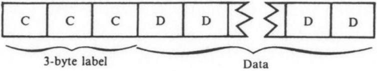


Fig. 9.10: Typische lijst "elementen" in het geheugen

DATA VOORSTELLING VOOR DE LIJST

De gewone en de alfabetische lijst gebruiken de zelfde voorstelling voor een lijst element:



Ieder element heeft een 3-bytes label, en een n-bytes blok data, met n tussen 1 en 253. Ieder element gebruikt dus ten hoogste 1 pagina (256 bytes). Alle elementen binnen een lijst hebben de zelfde lengte (zie figuur 9.10). De programma's van deze twee eenvoudige lijsten gebruiken de zelfde variabelen:

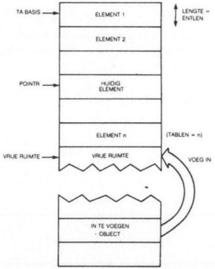


Fig. 9.11: De gewone lijst

ENTLEN is de lengte van een element. Voor een element met 10 bytes data, bijvoorbeeld, geldt: ENTLEN = 3 + 10 = 13
TABASE is de basis van de tabel of lijst in het geheugen
POINTR is de pointer naar het huidige element
OBJECT is het element waarnaar gezocht wordt, dat toegevoegd moet worden, of dat weggehaald moet worden.
TABLEN is het aantal elementen

Aangenomen wordt, dat alle tabels verschillend zijn. Er moet een kleine verandering in de programma's worden aangebracht, als van deze afspraak wordt afgeweken.

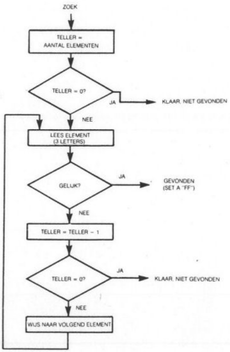


Fig. 9.12: Stroomdiagram. Zoeken in een tabel.

EEN GEWONE LIJST

Een gewone lijst is georganiseerd als een tabel met n elementen. Deze elementen zijn niet gesorteerd (zie figuur 9.11). Bij zoeken moet de hele lijst worden doorgewerkt, tot het element gevonden is, of tot het einde van de lijst bereikt is. Bij tussenvoegen (of toevoegen) worden de nieuwe ingangen toegevoegd aan de bestaande ingangen. Wordt een ingang gewist (weggehaald), dan worden de ingangen op hogere adressen naar beneden geschoven, om de tabel continu te houden.

Zoeken

Er wordt een seriele zoek techniek gebruikt. Iedere ingang label wordt vergeleken met de OBJECT label, karakter voor karakter.

De pointer POINTR wordt geinitialiseerd met de waarde van TABASE.

Het index register X wordt geinitialiseerd met het aantal elementen de lijst (opgeslagen in TABLEN).

Het zoeken gebeurt op de voor de hand liggende manier, en het stroomdiagram ervan is te vinden in figuur 9.12. Het programma staat

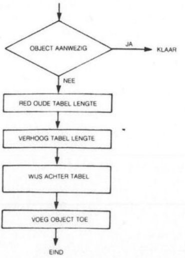


Fig. 9.13: Stroomdiagram. Tussenvoegen in een tabel.

in figuur 9.16 (het programma "SEARCH"). Figuur 9.17 geeft een voorbeeld van hoe het programma doorlopen wordt.

Tussenvoegen

Als een nieuw element wordt tussengevoegd, wordt het eerste beschikbare geheugen blok van (ENTLEN) bytes aan het eind van de tabel gebruikt (zie figuur 9.11).

Het programma test eerst of het nieuwe element al niet in de lijst voorkomt (alle labels zijn verschillend). Is dat niet het geval, dan wordt de lengte van de lijst, ENTLEN, met 1 verhoogd, en OBJECT wordt naar het eind van de lijst verplaatst. Dit gebeurt volgens het stroomdiagram van figuur 9.13.

Het programma staat in figuur 9.16. Het heet "NEW" en staat op adres 0135 tot 015E.

Het index register IY wijst naar de bron. HL en DE zijn de pointers naar de bestemming.

Wissen

Bij het wissen van een element worden alle volgende elementen op hogere adressen 1 element omhoog geschoven. De lengte van de lijst wordt verlaagd. Zie figuur 9.14.

Het programma is recht-toe-recht-aan, en is te vinden in figuur 9.16. Het heet "DELETE", en staat op adres 015F tot 0187. Figuur 9.15 geeft het stroomdiagram.

Het geheugen adres TEMPTR wordt gebruikt als een tijdelijke pointer naar het omhoog te schuiven element.

POINTR wijst daarbij naar het gat in de lijst, d.i. de bestemming van de volgende blok verplaatsing.

De Z vlag wordt gebruikt om aan te geven dat het wissen is geslaagd.

Merk op, dat DJNZ wordt gebruikt voor een efficiente blok verplaatsing (adres 0178 in figuur 9.16).

```
         LD    A, B              BLOK TELLER
NEWBLOC  LD    BC, (ENTLEN)      BLOK LENGTE
         LDIR
         DEC   A
         JP    NZ, NEWBLOC
```

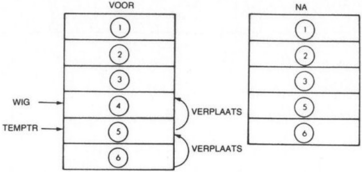


Fig. 9.14: Wissen van een element (gewone lijst)

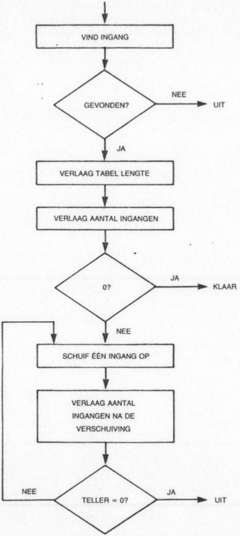


Fig. 9.15: Stroomdiagram. Wissen in een tabel

```
0000                            ORG     0100H
        (0187)          ENTLEN  DL      ENDER
        (0189)          TABLEN  DL      ENDER+2
        (018A)          TABASE  DL      ENDER+3
        (018C)          TEMP    DL      ENDER+5
                        ;
0100    1600            SEARCH  LD      D,0             ;CLEAR D
0102    3A8901                  LD      A,(TABLEN)      ;CHECK FOR A ZERO TABLE LENGTH
0105    A7                      AND     A               ;SET FLAGS
0106    C8                      RET     Z
0107    47                      LD      B,A             ;STORE TABLE LENGTH
0108    DD2A8A01                LD      IX,(TABASE)     ;PUT BASE ADDR. IN IX
010C    DD7E00          LOOP    LD      A,(IX+0)        ;CHECK FIRST LETTER OF ENTRY
010F    FDBE00                  CP      (IY+0)
0112    C22701                  JP      NZ,NEXTONE
0115    DD7E01                  LD      A,(IX+1)        ;CHECK 2ND LETTER
0118    FDBE01                  CP      (IY+1)
011B    C22701                  JP      NZ,NEXTONE
011E    DD7E02                  LD      A,(IX+2)        ;CHECK 3RD LETTER
0121    FDBE02                  CP      (IY+2)
0124    CA3201                  JP      Z,FOUND         ;EXIT IF ALL LETTERS MATCH
0127    05              NEXTONE DEC     B               ;DECREMENT TABLE LENGTH COUNTER
0128    C8                      RET     Z               ;EXIT IF AT END OF TABLE
0129    ED5B8701                LD      DE,(ENTLEN)     ;SET IX TO NEXT ENTRY ADDR.
012D    DD19                    ADD     IX,DE
012F    C30C01                  JP      LOOP            ;TRY AGAIN
0132    16FF            FOUND   LD      D,0FFH          ;SET D TO SHOW IX CONTAINS ADDR.
0134    C9                      RET                     ;..OF ENTRY IN TABLE
                        ;
                        ;
                        ;
0135    CD0001          NEW     CALL    SEARCH          ;SEE IF OBJECT IS THERE
0138    14                      INC     D
0139    CA5E01                  JP      Z,OUTE          ;IF D WAS FF, EXIT
013C    3A8901                  LD      A,(TABLEN)
013F    5F                      LD      E,A             ;LOAD E WITH TABLE LENGTH
0140    3C                      INC     A
0141    328901                  LD      (TABLEN),A      ;INCREMENT TABLE LENGTH
0144    1600                    LD      D,0
0146    2A8A01                  LD      HL,(TABASE)
0149    ED4B8701                LD      BC,(ENTLEN)     ;SET B TO LENGTH OF AN ENTRY
014D    41                      LD      B,C
014E    19              LOOPE   ADD     HL,DE
014F    10FD                    DJNZ    LOOPE           ;ADD HL TO (ENTLEN::TABLEN)
0151    ED4B8701                LD      BC,(ENTLEN)
0155    FDE5                    PUSH    IY              ;MOVE IY TO DE
0157    D1                      POP     DE
0158    EB                      EX      DE,HL
0159    EDB0                    LDIR                    ;MOVE MEMORY FROM OBJECT TO END
015B    01FFFF                  LD      BC,0FFFFH       ;..OF TABLE
015E    C9              OUTE    RET
                        ;
        .               ;
                        ;
015F    CD0001          DELETE  CALL    SEARCH          ;FIND ENTRY TO BE DELETED
0162    14                      INC     D               ;SEE IF IT WAS FOUND
0163    C28601                  JP      NZ,OUT
0166    3A8901                  LD      A,(TABLEN)      ;DECREMENT TABLE LENGTH
0169    3D                      DEC     A
016A    328901                  LD      (TABLEN),A
016D    05                      DEC     B               ;B NOW=# OF ENTRIES LEFT IN TABLE
016E    CA8301                  JP      Z,EXIT          ;..AFTER ONE TO BE DELETED
0171    DDE5                    PUSH    IX              ;MOVE IX TO DE
0173    D1                      POP     DE
0174    2A8701                  LD      HL,(ENTLEN)     ;SET HL ONE ENTRY AHEAD OF DE
0177    19                      ADD     HL,DE
0178    78                      LD      A,B             ;SET BLOCK COUNTER
0179    ED4B8701        NEWBLOC LD      BC,(ENTLEN)     ;SET BLOCK LENGTH COUNTER
017D    EDB0                    LDIR                    ;SHIFT 1 ENTRY OF TABLE
017F    3D                      DEC     A
0180    C27901                  JP      NZ,NEWBLOC      ;SHIFT ANOTHER BLOCK
0183    01FFFF          EXIT    LD      BC,0FFFFH       ;SHOW THAT IT WAS DONE
0186    C9              OUT     RET
                        ;
0187    (0000)          ENDER   END
```

Fig. 9.16: Gewone lijst. De programma's.

```
SYMBOL TABLE

DELETE  015F    ENDER   0187    ENTLEN  0187    EXIT    0183    FOUND   0132
LOOP    010C    LOOPE   014E    NEW     0135    NEWBLO  0179    NEXTON  0127
OUT     0186    OUTE    015E    SEARCH  0100    TABASE  018A    TABLEN  0189
TEMP    018C
```

Fig. 9.16: Gewone lijst. De programma's (vervolg)

Laat geheugen zien

Listing van objecten met hun adressen in geheugen

```
 DM300
0300  53 4F 4E 31 31 31 31 31-31 31 31 31 31 00 00 00  SON1111111111...
0310  44 41 44 32 32 32 32 32-32 32 32 32 32 00 00 00  DAD2222222222...
0320  4D 4F 4D 33 33 33 33 33-33 33 33 33 33 00 00 00  MOM3333333333...
0330  55 4E 43 34 34 34 34 34-34 34 34 34 34 00 00 00  UNC4444444444...
0340  41 4E 54 35 35 35 35 35-35 35 35 35 35 00 00 00  ANT5555555555...
0350  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0360  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0370  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................

-SY
Y=0000 300
```

maak 14 0300H (pointer naar OBJECT)

```
-G193/196

P=0196 0196'
```

Run 'INSERT'

Tabel na "insert"

```
-DM400
0400  53 4F 4E 31 31 31 31 31-31 31 31 31 31 00 00 00  SON1111111111...
0410  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0420  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0430  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0440  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0450  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0460  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0470  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................

-SY
Y=0300 310
```

maak 14 0310H (volgende OBJECT)

```
-G193/196

P=0196 0196'
```

Run 'INSERT'

tabel na 2de "insert"

```
-DM400
0400  53 4F 4E 31 31 31 31 31-31 31 31 31 31 44 41 44  SON1111111111DAD
0410  32 32 32 32 32 32 32 32-32 32 00 00 00 00 00 00  2222222222......
0420  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0430  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0440  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0450  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0460  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0470  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

• • • (meer toevoegingen) • • •

TABEL

```
-DM400
0400  53 4F 4E 31 31 31 31 31-31 31 31 31 31 44 41 44  SON1111111111DAD
0410  32 32 32 32 32 32 32 32-32 32 55 4E 43 34 34 34  2222222222UNC444
0420  34 34 34 34 34 34 34 4D-4F 4D 33 33 33 33 33 33  4444444MOM333333
0430  33 33 33 33 41 4E 54 35-35 35 35 35 35 35 35 35  3333ANT555555555
0440  35 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  5...............
0450  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0460  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0470  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

Fig. 9.17: Gewone lijst. Een voorbeeld "run"

```
-SY
Y=0340 320
-G190/193

P=0193 0193'   Run 'SEARCH'

                          Reg. D laat zien, dat object is gevonden

                                                                  Register inhouden
-DR
 Z  N   A=4D BC=02FF DE=FF0D HL=034D S=0100 P=0193 0193' CALL 0135
       A'=00 B'=0000 D'=0000 H'=0000 X=0427 Y=0320 I=00        (0135')

                                              Adres van object

-G196/199

P=0199 0199'   Run 'DELETE'                                    Tabel na "delete"

-DM400
0400  53 4F 4E 31 31 31 31 31-31 31 31 31 31 44 41 44   SON1111111111DAD
0410  32 32 32 32 32 32 32 32-32 32 55 4E 43 34 34 34   2222222222UNC444
0420  34 34 34 34 34 34 34 41-4E 54 35 35 35 35 35 35   4444444ANT555555
0430  35 35 35 35 41 4E 54 35-35 35 35 35 35 35 35 35   5555ANT555555555
0440  35 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   5...............
0450  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0460  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0470  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................

-SY
Y=0240 340
-G196/199      }  Wis laatste ingang uit tabel

P=0199 0199'                                              NB: schijnbaar
                                                          geen verandering
-DM400
0400  53 4F 4E 31 31 31 31 31-31 31 31 31 31 44 41 44   SON1111111111DAD
0410  32 32 32 32 32 32 32 32-32 32 55 4E 43 34 34 34   2222222222UNC444
0420  34 34 34 34 34 34 34 41-4E 54 35 35 35 35 35 35   4444444ANT555555
0430  35 35 35 35 41 4E 54 35-35 35 35 35 35 35 35 35   5555ANT555555555
0440  35 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   5...............
0450  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0460  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0470  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................

-DM189S1
0189  03  <---- Adres "tablen" - laat lengte tabel zien
-G190/193

P=0193 0193'    Run 'SEARCH' naar gewist object

                          D laat zien dat object niet gevonden is
-DR
 Z  N   A=55 BC=00FF DE=000D HL=0441 S=0100 P=0193 0193' CALL 0135
       A'=00 B'=0000 D'=0000 H'=0000 X=041A Y=0340 I=00        (0135')
```

Fig. 9.17: Gewone lijst. Een voorbeeld "run"

ALFABETISCHE LIJST

De alfabetische lijst, of tabel, is, in tegenstelling tot de vorige lijst, gesorteerd in alfabetische volgorde. Hierdoor is sneller zoeken mogelijk. We bekijken hier het binaire zoeken.

Zoeken

Het zoek algoritme gaat volgens een klassieke binaire zoek techniek. Deze techniek is in principe gelijk aan de manier waarop een naam in een telefoon boek opgezocht wordt. We beginnen ergens in het midden, en afhankelijk van wat we daar vinden, gaan we vooruit of achteruit om onze naam te vinden. Deze methode is snel en redelijk eenvoudig uit te voeren.

Het stroomdiagram staat in figuur 9.18, en het programma in figuur 9.23.

De tweede vlag, die gebruikt wordt door dit programma is CLOSE. Deze vlag wordt gelijk aan COMPRES als INCMNT gelijk aan 1 wordt. Het detecteert het feit, dat het element niet gevonden is, als COMPRES niet gelijk is aan CLOSE.

Andere programma variabelen zijn:

LOGPOS geeft de logische positie in de tabel (element nummer)

INCMNT geeft de waarde, waarmee de wijzer wordt verhoogd of verlaagd, als de volgende vergelijking niet slaagt

TABLEN is de lengte van de lijst

LOGPOS en INCMNT zullen vergeleken worden met TABLEN, om er zeker van te zijn, dat de grenzen van de tabel niet overschreden worden.

Het programma heet "SEARCH", en wordt in figuur 9.23 getoond. Het staat op adres 0100 tot 01CF. Dit programma moet zorgvuldig worden bestudeerd, want het is veel ingewikkelder dan het lineaire zoek programma.

Er kan een probleem optreden als gevolg van het feit dat een zoek interval even of oneven kan zijn. Is het oneven, dan moet er een correctie worden toegepast. (Er kan, bijvoorbeeld, niet gewezen worden naar het midden van een tabel met vier elementen.)

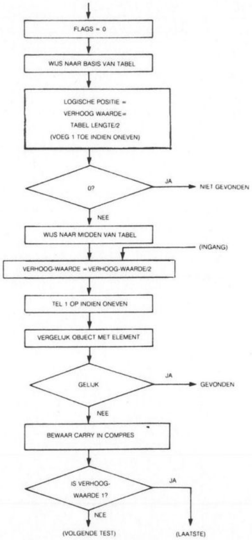


Fig. 9.18: Stroomdiagram. Binair zoeken.

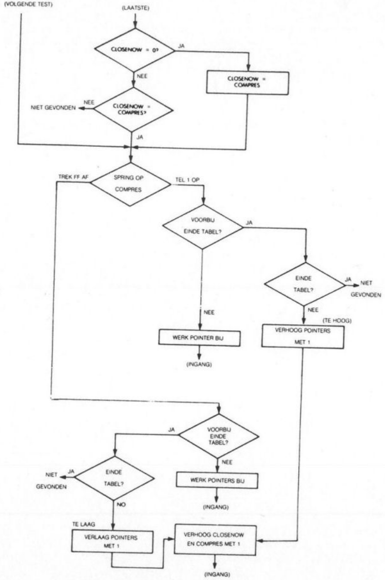


Fig 9.18: Stroomdiagram Binair Zoeken (vervolg)

Er moet in dat geval een "truc" worden toegepast. Er wordt door 2 gedeeld door naar rechts te schuiven. Is het bit dat in de carry schuift na de SRL instructie een 1, dan was het interval oneven. Het bit wordt dan bij de pointer opgeteld.

```
(0121)    LD     A,  C
          SRL    A
          ADC    0
          LD     C,  A
```

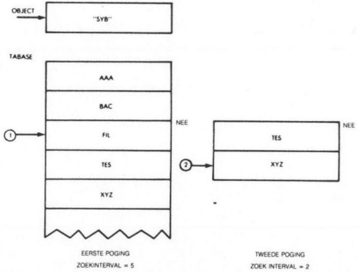


Fig. 9.19: Binair zoeken

OBJECT wordt vergeleken met het element in het midden van het interval. Is de vergelijking succesvol, dan is het programma klaar. Anders ("NOGOOD") wordt de carry 0, als OBJECT kleiner is dan het element. Wanneer INCMNT 1 wordt, wordt de CLOSE vlag (die oorspronkelijk 0 is gemaakt) gecontroleerd of deze geset is. Is dat niet het geval, dan wordt dat alsnog gedaan. Was de vlag geset, dan wordt gekeken waar OBJECT had moeten staan, maar waar het niet te vinden is.

De lijst bevat de elementen in alfabetische volgorde, en vindt een element d.m.v. een binaire of "logaritmische" zoek techniek. Een voorbeeld is in figuur 9.19 te vinden. Het zoeken wordt wat ingewikkeld gemaakt, doordat verschillende toestanden bijgehouden moeten worden. Het grootste probleem is, dat voorkomen moet worden, dat naar een niet bestaand element gezocht wordt. In dat geval is het mogelijk dat het programma blijft zoeken vlak boven en beneden de plaats waar het element had moeten staan. Om dat te voorkomen, wordt een vlag gebruikt in het programma, die de waarde van het carry bit bewaart na een niet geslaagde vergelijking. Als INCMNT, die aangeeft met welke waarde de pointer verhoogd moet worden, de waarde 1 krijgt, wordt een andere vlag, "CLOSENOW" (die we verder zullen afkorten tot CLOSE), gelijk gemaakt aan de waarde van de COMPRES vlag. Aldus zal COMPRES niet langer gelijk zijn aan CLOSE (omdat alle verhogingen met 1 gaan, gaat de pointer voorbij het punt, waar het gezochte element moet zijn), en het zoeken zal worden afgesloten. De NEW routine kan daarvan gebruik maken, om uit te zoeken, waar de logische en fysische wijzer staan t.o.v. de plaats waar het element heen zal gaan.

Als dus het OBJECT, waar naar gezocht wordt, niet in de tabel staat, en de pointer wordt met 1 verhoogd, dan wordt de CLOSE vlag geset. Tijdens de volgende doorloop van het programma zal het resultaat van de vergelijking tegengesteld zijn aan het vorige. De twee vlaggen zijn niet langer gelijk, en het programma zal beeindigd worden met de mededeling "niet gevonden".

Een ander belangrijk probleem is de mogelijkheid, dat de grenzen van de tabel worden overschreden bij het optellen of aftrekken van een waarde bij of van een pointer. Dit probleem wordt opgelost door een "proef" optelling of aftrekking uit te voeren. Daarbij wordt gebruik gemaakt van de logische pointer en de lengte, welke het huidige aantal elementen bijhoudt, en er wordt niet gebruik gemaakt van de werkelijke, fysieke geheugen adressen en pointers.

Samenvattend geldt dus, dat er twee pointers worden gebruikt om in-

formatie te onthouden: COMPRES en CLOSE. De COMPRES vlag onthoudt of de carry 0 of 1 is na de meest recente vergelijking. Het geeft aan, of het geteste element kleiner of groter is dan het gezochte element. Carry geeft de relatie aan. Is C 1, en is het element kleiner dan het object, dan wordt COMPRES 1 geset. Als carry 0 is, het element is dan groter dan het object, dan wordt COMPRES gelijk aan FF. Merk op, dat als carry 1 was (niet geslaagd), de wijzer naar de ingang onder OBJECT wijst.

Is carry niet geset, dan wordt dat alsnog gedaan. Is de carry geset, dan wordt onderzocht, of we de plaats waar OBJECT moet staan, maar niet stond, gepasseerd zijn.

Toevoegen van elementen

Om een element toe te kunnen voegen, wordt eerst binair gezocht in de tabel. Staat het element al in de tabel, dan hoeft het niet meer te worden toegevoegd. (We nemen aan, dat alle labels verschillend zijn.) Wordt het element niet gevonden, dan moet het toegevoegd worden direct voor of achter het laatste element, waarmee het vergeleken is. Compres geeft aan of het toevoegen voor of achter het laatste element zal gebeuren. Alle elementen volgend op het adres van het nieuwe element worden een blok positie naar beneden geschoven.

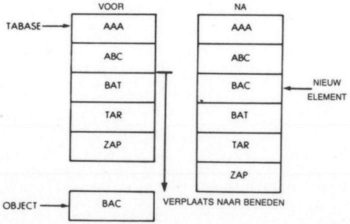


Fig. 9.20: Voeg toe: "BAC"

Dit proces wordt in figuur 9.20 geillustreerd, en het corresponderende programma staat in figuur 9.23.

Het programma heet "NEW", en start op adres 01D0. LDDR en LDIR worden gebruikt voor efficiente blok verplaatsingen.

Wissen van een element

Ook hier wordt binair gezocht naar het bewuste element. Is het niet aanwezig in de tabel, dan hoeft het niet te worden gewist. Slaagt het zoeken, dan wordt het element gewist, en alle volgende elementen worden een blok omhoog geschoven. Een voorbeeld staat in figuur 9.21, het programma in figuur 9.23, en het stroomdiagram in figuur 9.22.

Het programma heet "DELETE" en begint op adres 0221.

Een voorbeeld run staat in figuur 9.24. (Een programma uitvoeren is in het engels: to run a program. Vandaar de term "run".)

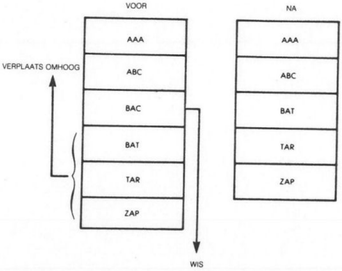


Fig. 9.21: Wis: "BAC"

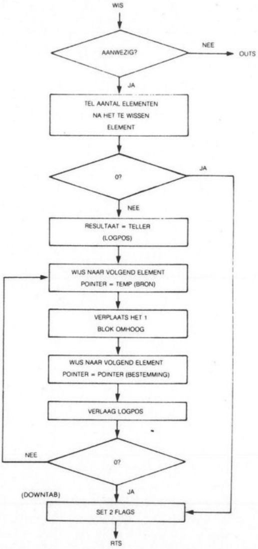


Fig. 9.22: Stroomdiagram. Wissen (alfabetische lijst).

```
0000                          ORG    0100H
        (024A)   CLOSENOW DL     ENDED
        (024B)   COMPRES  DL     ENDED+1
        (024C)   TABLEN   DL     ENDED+2
        (024D)   TABASE   DL     ENDED+3
        (024F)   ENTLEN   DL     ENDED+5
                 ;
0100    3E00     SEARCH   LD     A,0
0102    324A02            LD     (CLOSENOW),A    ;ZERO FLAG LOCATIONS
0105    324B02            LD     (COMPRES),A
0108    57                LD.    D,A
0109    2A4D02            LD     HL,(TABASE)     ;INITIALIZE HL
010C    3A4C02            LD.    A,(TABLEN)
010F    CB3F              SRL    A               ;DIVIDE BY 2
0111    CE00              ADC    0               ;ADD 1'S BIT BACK IN
0113    4F                LD     C,A             ;STORE AS INCREMENT VALUE
0114    47                LD     B,A             ;STORE AS LOGICAL POSITION VALUE
0115    CABA01            JP     Z,NOTFOUND      ;CHECK IF LENGTH IS ZERO
0118    5F                LD     E,A             ;MULTIPLY (E-1)::ENTLEN
0119    1D                DEC    E
011A    CDBD01            CALL   MULT
011D    19                ADD    HL,DE           ;SET HL TO MIDDLE OF TABLE
011E    E5       ENTRY    PUSH   HL              ;LOAD HL INTO IX
011F    DDE1              POP    IX
0121    79                LD     A,C             ;DIVIDE INCREMENT VALUE BY TWO
0122    CB3F              SRL    A
0124    CE00              ADC    0
0126    4F                LD     C,A
0127    DD7E00            LD     A,(IX+0)        ;COMPARE FIRST LETTER
012A    FDBE00            CP     (IY+0).
012D    C24201            JP     NZ,NOGOOD
0130    DD7E01            LD     A,(IX+1)        ;COMPARE 2ND LETTER
0133    FDBE01            CP     (IY+1)
0136    C24201            JP     NZ,NOGOOD
0139    DD7E02            LD     A,(IX+2)        ;COMPARE 3RD LETTER
013C    FDBE02            CP     (IY+2)
013F    CABC01            JP     Z,FOUND
0142    3E01     NOGOOD   LD     A,1             ;SET COMPARE RESULT FLAG TO
0144    DA4901            JP     C,TESTS         ;..RESULT OF COMPARE (1,FF)
0147    3EFF              LD     A,0FFH
0149    324B02   TESTS    LD     (COMPRES),A
014C    79                LD     A,C             ;IS INCREMENT VALUE 1?
014D    3D                DEC    A
014E    C26901            JP     NZ,NEXTEST
0151    3A4A02            LD     A,(CLOSENOW)    ;YES, IS CLOSE FLAG SET?
0154    A7                AND    A
0155    CA6301            JP     Z,NOTCLOSE
0158    57                LD     D,A             ;YES,SEE IF HAVE PASSED WHERE
0159    3A4B02            LD     A,(COMPRES)     ;..ENTRY SHOULD BE BUT ISN'T
015C    92                SUB    D
015D    CA6901            JP     Z,NEXTEST
0160    C3BA01            JP     NOTFOUND
0163    3A4B02   NOTCLOSE LD     A,(COMPRES)     ;SET CLOSE FLAG TO DIRECTION OF
0166    324A02            LD     (CLOSENOW),A    ;..SEARCH TO PREVENT REPETITION
0169    DDE5     NEXTEST  PUSH   IX              ;PREPARE HL AND DE FOR ADD OR
016B    E1                POP    HL              ;..SUB OF INCREMENT VALUE
016C    59                LD     E,C
016D    CDBD01            CALL   MULT
0170    3A4B02            LD     A,(COMPRES)     ;TEST IF WANT TO ADD OR SUB
0173    3C                INC    A
0174    C29601            JP     NZ,ADDIT
0177    78                LD     A,B             ;TEST TO SEE IF SUB WILL RUN
0178    91                SUB    C               ;..OFF BOTTOM OF TABLE
0179    CA8501            JP     Z,TOOLOW
017C    DA8501            JP     C,TOOLOW
017F    47                LD     B,A             ;SET NEW LOGICAL POSITION VALUE
0180    ED52              SBC    HL,DE           ;CHANGE ADDRESS ITSELF
0182    C31E01            JP     ENTRY
0185    78       TOOLOW   LD     A,B             ;SEE IF POSITION IS 1
0186    3D                DEC    A
0187    CABA01            JP     Z,NOTFOUND      ;IF SO, EXIT
018A    ED5B4F02          LD     DE,(ENTLEN)     ;JUST SUB 1 ENTRY POSITION
018E    37                SCF
018F    3F                CCF
0190    ED52              SBC    HL,DE
0192    05                DEC    B               ;CHANGE LOGICAL POSITION
0193    C3AF01            JP     REALCLOS
```

Fig. 9.23: Binair zoek programma

```
0196  3A4C02    ADDIT    LD    A,(TABLEN)      ;TEST TO SEE IF CURRENT POSITION
0199  90                 SUB   B               ;..PLUS INCREMENT WILL GO PAST
019A  91                 SUB   C               ;...END OF THE TABLE
019B  DAA501             JP    C,TOOHIGH
019E  19                 ADD   HL,DE           ;IS OK, CHANGE ACTUAL ADDRESS
019F  78                 LD    A,B             ;CHANGE LOGICAL POS. VALUE
01A0  81                 ADD   C
01A1  47                 LD    B,A
01A2  C31E01             JP    ENTRY
01A5  81        TOOHIGH  ADD   C               ;SEE IF POSITION IS AT TOP OF
01A6  CABA01             JP    Z,NOTFOUND      ;..TABLE (SAME AS TABLEN B)
01A9  ED5B4F02           LD    DE,(ENTLEN)     ;ADD 1 ENTRY POSITION
01AD  19                 ADD   HL,DE
01AE  04                 INC   B               ;INCREMENT LOGICAL POSITION
01AF  0E01      REALCLOS LD    C,1             ;SET INCREMENT TO 1
01B1  3A4B02             LD    A,(COMPRES)     ;SET CLOSE FLAG TO COMPARE
01B4  324A02             LD    (CLOSENOW),A    ;..RESULT
01B7  C31E01             JP    ENTRY
01BA  16FF      NOTFOUND LD    D,0FFH
01BC  C9        FOUND    RET
                ;
01BD  E5        MULT     PUSH  HL              ;MULTIPLIES E BY (ENTLEN),
01BE  C5                 PUSH  BC              ;..VALUE IN DE ON EXIT
01BF  1600               LD    D,0
01C1  210000             LD    HL,0000
01C4  ED4B4F02           LD    BC,(ENTLEN)
01C8  41                 LD    B,C
01C9  19        ADDEM    ADD   HL,DE
01CA  10FD               DJNZ  ADDEM
01CC  C1                 POP   BC
01CD  EB                 EX    DE,HL
01CE  E1                 POP   HL
01CF  C9                 RET
                ;
                ;
                ;
01D0  CD0001    NEW      CALL  SEARCH          ;SEE IF OBJECT IS ALREADY THERE
01D3  14                 INC   D
01D4  C22002             JP    NZ,OUT
01D7  3A4C02             LD    A,(TABLEN)      ;CHECK FOR 0 TABLE
01DA  A7                 AND   A
01DB  CA0C02             JP    Z,INSERT
01DE  3A4B02             LD    A,(COMPRES)
01E1  3C                 INC   A
01E2  CAED01             JP    Z,HISIDE
01E5  ED5B4F02           LD    DE,(ENTLEN)     ;COMPRES=1, SET HL ABOVE WHERE
01E9  19                 ADD   HL,DE           ;..OBJECT SHOULD GO
01EA  C3EE01             JP    SETUP
01ED  05        HISIDE   DEC   B               ;COMPRES=0, SET B FOR SUBTRACT
01EE  3A4C02    SETUP    LD    A,(TABLEN)      ;SEE HOW MANY ENTRES ARE LEFT
01F1  90                 SUB   B
01F2  CA0C02             JP    Z,INSERT
01F5  5F                 LD    E,A             ;SET HL TO LAST POSITION IN LAST
01F6  CDBD01             CALL  MULT            ;..ENTRY
01F9  19                 ADD   HL,DE
01FA  2B                 DEC   HL
01FB  EB                 EX    DE,HL           ;SET DE 1 ENTRY ABOVE HL
01FC  2A4F02             LD    HL,(ENTLEN)
01FF  19                 ADD   HL,DE
0200  EB                 EX    DE,HL
0201  ED4B4F02  MOVEM    LD    BC,(ENTLEN)     ;SHIFT UP ONE ENTRY OF MEMORY
0205  EDB8               LDDR
0207  3D                 DEC   A
0208  C20102             JP    NZ,MOVEM        ;REPEAT IF NECCESSARY
020B  23                 INC   HL              ;HL IS FRONT OF NOW EMPTY SPACE
020C  FDE5      INSERT   PUSH  IY              ;LOAD OBJECT INTO EMPTY SPACE
020E  D1                 POP   DE
020F  EB                 EX    DE,HL
0210  ED4B4F02           LD    BC,(ENTLEN)
0214  EDB0               LDIR
0216  3A4C02             LD    A,(TABLEN)      ;INCREMENT TABLE LENGTH
0219  3C                 INC   A
021A  324C02             LD    (TABLEN),A
021D  01FFFF             LD    BC,0FFFFH       ;SHOW THAT IT WAS DONE
0220  C9        OUT      RET
                ;
                ;
                ;
```

Fig. 9.23: Binair zoek programma (vervolg)

```
0221  CD0001    DELETE   CALL  SEARCH        ;GET ADDRESS OF OBJECT
0224  14                 INC   D             ;SEE IF OBJECT IS THERE
0225  CA4902             JP    Z,OUTE
0228  ED5B4F02           LD    DE,(ENTLEN)
022C  EB                 EX    DE,HL
022D  19                 ADD   HL,DE         ;DE IS LOC. OF OBJECT, HL IS
022E  3A4C02             LD    A,(TABLEN)    ;..ONE ENTRY ABOVE
0231  90                 SUB   B             ;SEE HOW MANY ENTRIES ARE LEFT
0232  CA3F02             JP    Z,DOWNTAB
0235  ED4B4F02  SHIFTIN  LD    BC,(ENTLEN)
0239  EDB0               LDIR                ;SHIFT DOWN 1 ENTRY LENGTH
023B  3D                 DEC   A
023C  C23502             JP    NZ,SHIFTIN
023F  3A4C02    DOWNTAB  LD    A,(TABLEN)    ;DECREMENT TABLE LENGTH
0242  3D                 DEC   A
0243  324C02             LD    (TABLEN),A
0246  01FFFF             LD    BC,OFFFFH     ;SHOW THAT ACTION WAS TAKEN
0249  C9        OUTE     RET
                ;
024A  (0000)    ENDED    END

SYMBOL TABLE

ADDEM   01C9   ADDIT   0196   CLOSEN  024A   COMPRE  024B   DELETE  0221
DOWNTA  023F   ENDED   024A   ENTLEN  024F   ENTRY   011E   FOUND   01BC
HISIDE  01ED   INSERT  020C   MOVEM   0201   MULT    01BD   NEW     01D0
NEXTES  0169   NOGOOD  0142   NOTCLO  0163   NOTFOU  01BA   OUT     0220
OUTE    0249   REALCL  01AF   SEARCH  0100   SETUP   01EE   SHIFTI  0235
TABASE  024D   TABLEN  024C   TESTS   0149   TOOHIG  01A5   TOOLOW  0185
```

Fig. 9.23: Binair zoek programma (vervolg)

GEKETENDE LIJST

De geketende lijst bevat, zoals gewoonlijk, de drie alfa-numerieke karakters van het label, gevolgt door 1 tot 250 bytes data, gevolgt door een 2-bytes pointer met het start adres van het volgende element, tenslotte gevolgd door een 1-bytes merkteken. Is het merkteken een 1, dan voorkomt het, dat een nieuw element op de plaats van het oude wordt gezet.

Verder bevat een directory een pointer naar het eerste element voor iedere letter van het alfabet, dit om het opzoeken te vergemakkelijken. Aangenomen wordt, dat de labels ASCII karakters zijn. Alle pointers aan het eind van de lijst krijgen de zogenaamde NIL waarde, in dit geval is die waarde gelijk aan de basis van de tabel. Deze waarde komt nergens anders in de tabel voor.

Het toevoeg en het wis programma voeren de voor de hand liggende pointer manipulaties uit. De vlag INDEX wordt gebruikt om aan te geven, of een wijzer afkomstig is van een vorig element, of van de directory. De programma's staan in figuur 9.29.

Figuur 9.25 geeft de data structuur.

Oorspronkelijke tabel

```
DM400
0400   00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0410   00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0420   00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0430   00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0440   00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0450   00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0460   00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0470   00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
```

Listing van objecten in geheugen

```
-DM300
0300   53 4F 4E 31 31 31 31 31-31 31 31 31 31 00 00 00   SON1111111111...
0310   44 41 44 32 32 32 32 32-32 32 32 32 32 00 00 00   DAD2222222222...
0320   4D 4F 4D 33 33 33 33 33-33 33 33 33 33 00 00 00   MOM3333333333...
0330   55 4E 43 34 34 34 34 34-34 34 34 34 34 00 00 00   UNC4444444444...
0340   41 4E 54 35 35 35 35 35-35 35 35 35 35 00 00 00   ANT5555555555...
0350   00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0360   00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0370   00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
-SY
Y=0000 320
-G263/266

F=0266 0266'
```

Run 'INSERT'

Tabel na invoegen

```
-DM400
0400   4D 4F 4D 33 33 33 33 33-33 33 33 33 33 00 00 00   MOM3333333333...
0410   00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0420   00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0430   00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0440   00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0450   00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0460   00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0470   00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
-SY
Y=0320 310
-G263/266

F=0266 0266'
```

Run "INSERT" met ander object

Tabel na invoegen. NB: nog steeds alfabetisch.

```
-DM400
0400   44 41 44 32 32 32 32 32-32 32 32 32 32 4D 4F 4D   DAD2222222222MOM
0410   33 33 33 33 33 33 33 33-33 33 00 00 00 00 00 00   3333333333......
0420   00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0430   00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0440   00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0450   00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0460   00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0470   00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
```

• • •nog meer invoegingen• • •

Fig. 9.24: Alfabetische lijst. Een voorbeeld run

Tabel na invoegen

```
-DM400
0400  41 4E 54 35 35 35 35 35-35 35 35 35 35 44 41 44  ANT5555555555DAD
0410  32 32 32 32 32 32 32 32-32 32 4D 4F 4D 33 33 33  2222222222MOM333
0420  33 33 33 33 33 33 33 53-4F 4E 31 31 31 31 31 31  3333333SON111111
0430  31 31 31 31 55 4E 43 34-34 34 34 34 34 34 34 34  1111UNC444444444
0440  34 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  4...............
0450  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0460  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0470  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
-SY
Y=0340 300
-G260/263

P=0263 0263'
```

Run "SEARCH" voor "SON" (op adres 0300)

gevonden (DE=000D)

```
-DR
 Z  N   A=4E BC=0401 DE=000D HL=0427 S=0100 P=0263 0263' CALL 01D0
       A'=00 B'=0000 D'=0000 H'=0000 X=0427 Y=0300 I=00        (01D0')
```

Adres van Object in tabel (X=0427)

(verify in Tab (controleer dat het "SON" is)

```
-G266/269
P=0269 0269'
```

Run "DELETE" voor "SON"

Tabel na wissen. NB: UNC is opgeschoven, de laatste UNC moet genegeerd worden

```
-DM400
0400  41 4E 54 35 35 35 35 35-35 35 35 35 35 44 41 44  ANT5555555555DAD
0410  32 32 32 32 32 32 32 32-32 32 4D 4F 4D 33 33 33  2222222222MOM333
0420  33 33 33 33 33 33 33 55-4E 43 34 34 34 34 34 34  3333333UNC444444
0430  34 34 34 34 55 4E 43 34-34 34 34 34 34 34 34 34  4444UNC444444444
0440  34 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  4...............
0450  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0460  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0470  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................

-G260/263
P=0263 0263'
```

Probeer run of "SEARCH" nog eens (voor "SON")

Niet gevonden (DE=FF0D)

```
-DR
S   N   A=FE BC=0401 DE=FF0D HL=0427 S=0100 P=0263 0263' CALL 01D0
       A'=00 B'=0000 D'=0000 H'=0000 X=0427 Y=0300 I=00        (01D0')
-G263/266
P=0266 0266'
```

Voeg object weer in

Huidige tabel. Vergelijk deze met tabel voor "DELETE"

```
-DM400
0400  41 4E 54 35 35 35 35 35-35 35 35 35 35 44 41 44  ANT5555555555DAD
0410  32 32 32 32 32 32 32 32-32 32 4D 4F 4D 33 33 33  2222222222MOM333
0420  33 33 33 33 33 33 33 53-4F 4E 31 31 31 31 31 31  3333333SON111111
0430  31 31 31 31 55 4E 43 34-34 34 34 34 34 34 34 34  1111UNC444444444
0440  34 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  4...............
0450  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0460  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0470  00 00 00 00 00 00 00 00-00 00 00 00 00 00 G0 00  ................
```

Laat zien dat actie is uitgevoerd (BC=FFFF)

```
-DR
        A=05 BC=FFFF DE=0434 HL=030D S=0100 P=0266 0266' CALL 0221
       A'=00 B'=0000 D'=0000 H'=0000 X=0427 Y=0300 I=00        (0221')
```

Fig. 9.24: Alfabetische lijst. Een voorbeeld run (vervolg)

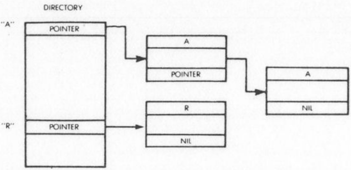
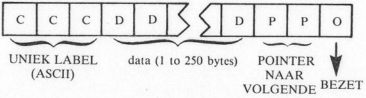


Fig. 9.25: Geketende lijst structuur

Een toepassing van deze data structuur is een adresboek in een computer. Iedere persoon wordt voorgesteld door een unieke 3-letter code (bijvoorbeeld de initialen), en het dataveld bevat een vereenvoudigd adres, plus het telefoon nummer (tot 250 karakters). Laten we de structuur eens nader bekijken in figuur 9.23. Het formaat van ieder element is:



De afspraken zijn (zoals gebruikelijk):

ENTLEN totale element lengte in bytes
TABASE basis adres van de lijst

Het adres van het OBJECT staat altijd in het IY register voordat het programma wordt uitgevoerd. REFBASE wijst naar de basis van de directory, of "referentie tabel".

Ieder twee-bytes adres in deze directory wijst naar het eerste element van de letter waarmee het correspondeert in de lijst. Iedere groep

elementen met de zelfde eerste letter in de label vormt in feite een aparte lijst in de hele structuur. Het maakt zoeken mogelijk en is analoog aan het adres boek. Merk op, dat de data niet wordt verplaatst tijdens wissen en tussenvoegen, maar dat alleen de pointers worden veranderd.

Wordt geen element beginnend met een bepaalde letter gevonden, of is er geen element alfabetisch volgend op een bestaand element, dan wijzen de pointers naar het begin van de tabel (= "NIL"). Op de bodem van de tabel wordt een getal opgeslagen, zodanig dat het absolute verschil tussen dat getal en "Z" groter is dan het absolute verschil tussen "A" en "Z". Dit getal stelt het "eind van de tabel" merkteken voor, EOT (afkorting van "end of table"). Er wordt aangenomen, dat EOT net zoveel geheugen in gebruik neemt als een normaal element, maar het zou ook 1 byte lang kunnen zijn, indien gewenst. De letters zijn alfabetische letters in ASCII code. Wil je dat veranderen, dan moet de constante in de PRETAB routine veranderd worden.

EOT heeft de waarde van het begin van de tabel (NIL).

Alle NIL pointers aan het eind van een reeks karakters of in een directory adres, dat nergens naar wijst, krijgen de waarde van de tabel basis, om een unieke identificatie te krijgen. Er kunnen ook andere afspraken gevolgd worden.

Toevoegen en wissen geschieden op de gebruikelijke manier (zie deel 1 van dit hoofdstuk), door de betreffende pointers te veranderen. De INDEXED vlag wordt gebruikt, om aan te geven of de pointer, naar het object, in de referentie tabel staat of in een ander element.

Zoeken

Het SEARCH (zoek) programma staat op adres 0100 tot 0155, en gebruikt de subroutine PRETAB op adres 01D2

Het zoek principe is eenvoudig:

1 - haal het directory element corresponderend met de letter van het alfabet in de eerste positie van de OBJECT label

2 - haal de pointer; kijk of het element NIL is; zoja, dan bestaat het gezochte element niet

3 - indien niet NIL, vergelijk het element met OBJECT; indien gelijk, dan is het zoeken klaar; indien niet gelijk, haal dan de volgende pointer

4 - ga terug naar 2

Figuur 9.26 geeft een voorbeeld.

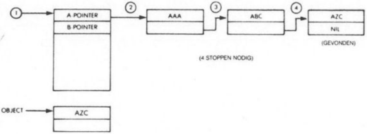


Fig. 9.26: Geketende lijst. Zoeken.

Toevoegen

Toevoegen is in principe zoeken, gevolgd door toevoegen als een NIL gevonden wordt.

Er wordt een geheugen blok voor het nieuwe element voorbij EOT gezocht, door te kijken naar een merkteken dat zegt "beschikbaar".

Het programma heet "NEW" in figuur 9.29, en staat op adres 0156 tot 01A3. Een voorbeeld is in figuur 9.27 te vinden.

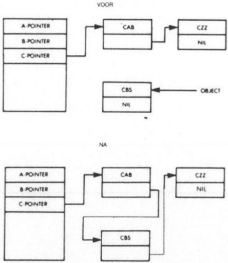


Fig. 9.27: Geketende lijst. Toevoegen.

Wissen

Het element wordt gewist door zijn merkteken "beschikbaar" te maken, en de pointer ernáar toe vanuit de directory of een ander element te veranderen.

Het programma heet "DELETE", en staat op adres 01A4 tot 01D1. Figuur 9.28 geeft een voorbeeld.

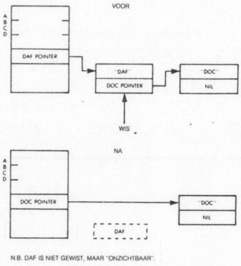


Fig. 9.28: Voorbeeld van wissen (geketende lijst)

```
0000'                           ORG     0100H
        (01F7)  INDEXFD  DL     ENDREF
        (01F8)  TABASE   DL     ENDREF+1
        (01FA)  RELBASE  DL     ENDREF+3
        (01FC)  ENTLEN   DL     ENDREF+5
                ;
0100    3E00    SEARCH   LD     A,0             ;INITIALIZE FLAGS
0102    47               LD     B,A
0103    3C               INC    A
0104    32F701           LD     (INDEXFD),A
0107    CDD201           CALL   PRETAB          ;GET ADDR OF INDEX POINTER
010A    1A               LD     A,(DE)          ;MOVE POINTER CONTENTS TO HL
010B    6F               LD     L,A
010C    13               INC    DE
010D    1A               LD     A,(DE)
010E    67               LD     H,A
010F    E5               PUSH   HL
0110    DDE1             POP    IX
0112    DD7E00  COMPARE  LD     A,(IX+0)        ;LOOK AT FIRST LETTER OF ENTRY
0115    FE2C             CP     2CH             ;SEE IF IS EOT MARKER
0117    D25501           JP     NC,NOTFOUND
011A    DD7E00           LD     A,(IX+0)        ;COMPARE FIRST LETTERS
011D    FDBE00           CP     (IY+0)
0120    DA3E01           JP     C,NOGOOD
0123    C25501           JP     NZ,NOTFOUND
0126    DD7E01           LD     A,(IX+1)        ;COMPARE 2ND LETTERS
0129    FDBE01           CP     (IY+1)
012C    DA3E01           JP     C,NOGOOD
012F    C25501           JP     NZ,NOTFOUND
0132    DD7E02           LD     A,(IX+2)        ;COMPARE 3RD LETTERS
0135    FDBE02           CP     (IY+2)
0138    CA5301           JP     Z,FOUND
013B    D25501           JP     NC,NOTFOUND
013E    DDE5    NOGOOD   PUSH   IX
0140    D1               POP    DE
0141    2AFC01           LD     HL,(ENTLEN)     ;JUMP TO POINTER OF ENTRY
0144    19               ADD    HL,DE
0145    4E               LD     C,(HL)          ;PUT POINTER VALUE IN BC
0146    23               INC    HL
0147    46               LD     B,(HL)
0148    C5               PUSH   BC              ;LOAD IX WITH POINTER
0149    DDE1             POP    IX
014B    3E00             LD     A,0
014D    32F701           LD     (INDEXFD),A     ;RESET FLAG
0150    C31201           JP     COMPARE
0153    06FF    FOUND    LD     B,0FFH
0155    C9      NOTFOUND RET
                ;
                ;
                ;
0156    CD0001  NEW      CALL   SEARCH          ;SEE WHERE OBJECT SHOULD GO
0159    04               INC    B
015A    CAA301           JP     Z,OUT
015D    D5               PUSH   DE              ;STORE ADDR. OF PREVIOUS ENTRY
015E    2AF801           LD     HL,(TABASE)     ;FIND SPACE IN TABLE FOR NEW
0161    EB      NEXTONE  EX     DE,HL           ;MOVE TO END OF NEXT ENTRY
0162    2AFC01           LD     HL,(ENTLEN)
0165    23               INC    HL              ;ADD 3 FOR REAL LENGTH OF ENTRY
0166    23               INC    HL
0167    23               INC    HL
0168    19               ADD    HL,DE
0169    7E               LD     A,(HL)
016A    3D               DEC    A
016B    CA6101           JP     Z,NEXTONE       ;IF SOMETHING IS THERE, TRY AGAIN
016E    13               INC    DE
016F    D5               PUSH   DE              ;SAVE POSITION OF EMPTY SPACE
0170    FDE5             PUSH   IY              ;MOVE IY TO HL
0172    E1               POP    HL
0173    ED4BFC01         LD     BC,(ENTLEN)     ;MOVE OBJECT INTO TABLE
0177    EDB0             LDIR
0179    DDE5             PUSH   IX              ;PUT ADDR OF ENTRY AFTER OBJECT
017B    E1               POP    HL              ;...AT POINTER POSITION
017C    EB               EX     DE,HL
017D    73               LD     (HL),E
017E    23               INC    HL
017F    72               LD     (HL),D
0180    23               INC    HL
0181    3601             LD     (HL),1          ;SET OCCUPANCY MARKER
```

Fig. 9.29: Geketende lijst. De programma's.

```
0183  E1                POP   HL           ;GET ADDR OF WHERE THIS SPACE IS
0184  3AE701            LD    A,(INDEXED)  ;SEE WHAT PREVIOUS POINTERS MUST
0187  3D                DEC   A            ;..BE SET
0188  CA9801            JP    Z,SETINX
018B  E3                EX    (SP),HL      ;GET ADDR OF ENTRY PREVIOUS TO
018C  ED5BEC01          LD    DE,(ENTLEN)  ;..OBJECT & MOVE TO POINTER AREA
0190  19                ADD   HL,DE
0191  D1                POP   DE           ;RETRIEVE ADDR OF OBJECT
0192  73                LD    (HL),E       ;PUT IT AT POINTER POSITION
0193  23                INC   HL
0194  72                LD    (HL),D
0195  C3A001            JP    FINISH
0198  C1        SETINX  POP   BC           ;CLEAR OUT STACK
0199  CDD201            CALL  PRETAB       ;GET INDEX ADDRESS
019C  EB                EX    DE,HL        ;LOAD HL INTO IT
019D  73                LD    (HL),E
019E  23                INC   HL
019F  72                LD    (HL),D
01A0  01FFFF    FINISH  LD    BC,0FFFFH    ;SHOW THAT IT WAS DONE
01A3  C9        OUT     RET
                ;
                ;
                ;
01A4  CD0001    DELETE  CALL  SEARCH       ;GET ADDRESS OF OBJECT
01A7  04                INC   B            ;SEE IF IT IS THERE
01A8  C2D101            JP    NZ,OUTE
01AB  DDE5              PUSH  IX           ;SET HL TO POINTER AREA OF OBJECT
01AD  E1                POP   HL
01AE  ED4BEC01          LD    BC,(ENTLEN)
01B2  09                ADD   HL,BC
01B3  4E                LD    C,(HL)       ;RETRIEVE POINTER
01B4  23                INC   HL
01B5  46                LD    B,(HL)
01B6  23                INC   HL
01B7  3600              LD    (HL),0       ;REMOVE OCCUPANCY MARKER
01B9  3AE701            LD    A,(INDEXED)  ;SEE IF INDEX NEEDS CHANGING
01BC  3D                DEC   A
01BD  C2C701            JP    NZ,CHANGEM
01C0  CDD201            CALL  PRETAB       ;YES,PUT ADDR INTO HL
01C3  EB                EX    DE,HL
01C4  C3CB01            JP    MOVIN
01C7  2AEC01    CHANGEM LD    HL,(ENTLEN)  ;SET HL TO POINTER OF PREVIOUS
01CA  19                ADD   HL,DE
01CB  71        MOVIN   LD    (HL),C       ;PUT ADDR OF NEXT INTO WHATEVER
01CC  23                INC   HL           ;..(EITHER INDEX OR ENTRY)
01CD  70                LD    (HL),B
01CF  01FFFF            LD    BC,0FFFFH
01D1  C9        OUTE    RET
                ;
                ;
                ;
01D2  E5        PRETAB  PUSH  HL
01D3  FD7E00            LD    A,(IY+0)     ;GET FIRST LETTER OF OBJECT
01D6  3D                DEC   A            ;REMOVE ASCII LEADER
01D7  D640              SUB   40H
01D9  CB27              SLA   A            ;MULTIPLY BY 2
01DB  2AEA01            LD    HL,(REFBASE)
01DE  85                ADD   L
01DF  6F                LD    L,A
01E0  D2E401            JP    NC,FIXUP
01E3  24                INC   H
01E4  EB        FIXUP   EX    DE,HL
01E5  E1                POP   HL
01E6  C9                RET
                ;
01E7  (0000)    ENDER   END
```

SYMBOL TABLE

CHANGE	01C7	COMPAR	0112	DELETE	01A4	ENDER	01E7	ENTLEN	01EC
FINISH	01A0	FIXUP	01E4	FOUND	0153	INDEXE	01E7	MOVIN	01CB
NEW	0156	NEXTON	0161	NOGOOD	013F	NOTFOU	0155	OUT	01A3
OUTE	01D1	PRETAB	01D2	REFBAS	01EA	SEARCH	0100	SETINX	0198
TABASE	01E8								

Fig. 9.29: Geketende lijst. De programma's. (vervolg)

De objecten in geheugen

Listing objecten en hun adressen in geheugen

```
DM300
0300   53 4F 4E 31 31 31 31 31 31 31 31 31 31 00 00 00   SON1111111111...
0310   44 41 44 32 32 32 32 32 32 32 32 32 32 00 00 00   DAD2222222222...
0320   4D 4F 4D 33 33 33 33 33 33 33 33 33 33 00 00 00   MOM3333333333...
0330   55 4E 43 34 34 34 34 34 34 34 34 34 34 00 00 00   UNC4444444444...
0340   41 4E 54 35 35 35 35 35 35 35 35 35 35 00 00 00   ANT5555555555...
0350   41 41 41 36 36 36 36 36 36 36 36 36 36 00 00 00   AAA6666666666...
0360   41 5A 5A 37 37 37 37 37 37 37 37 37 37 00 00 00   AZZ7777777777...
0370   53 49 44 38 38 38 38 38 38 38 38 38 38 00 00 00   SID8888888888...
```

EOT karakter in oorspronkelijke tabel

```
DM400
0400   7B 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   {...............
0410   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0420   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0430   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0440   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0450   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0460   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0470   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
```

Oorspronkelijke directory

```
-DM500
0500   00 04 00 04 00 04 00 04 00 04 00 04 00 04 00 04   ................
0510   00 04 00 04 00 04 00 04 00 04 00 04 00 04 00 04   ................
0520   00 04 00 04 00 04 00 04 00 04 00 04 00 04 00 04   ................
0530   00 04 00 04 00 00 00 00 00 00 00 00 00 00 00 00   ................
0540   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0550   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0560   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0570   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
```

Bezet tekens

Pointers

Tabel na verschillende invoegingen

```
DM400
0400   7B 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   {...............
0410   41 4E 54 35 35 35 35 35 35 35 35 35 35 70 04 01   ANT5555555555p..
0420   44 41 44 32 32 32 32 32 32 32 32 32 32 00 04 01   DAD2222222222...
0430   41 41 41 36 36 36 36 36 36 36 36 36 36 10 04 01   AAA6666666666...
0440   53 4F 4E 31 31 31 31 31 31 31 31 31 31 00 04 01   SON1111111111...
0450   4D 4F 4D 33 33 33 33 33 33 33 33 33 33 00 04 01   MOM3333333333...
0460   53 49 44 38 38 38 38 38 38 38 38 38 38 40 04 01   SID8888888888@..
0470   41 5A 5A 37 37 37 37 37 37 37 37 37 37 00 04 01   AZZ7777777777...
```

```
SY
Y 0460 316
6226/229

F 0229 0229'
```

Wis een element

Enige verandering

```
DM400
0400   7B 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   {...............
0410   41 4E 54 35 35 35 35 35 35 35 35 35 35 70 04 01   ANT5555555555p..
0420   44 41 44 32 32 32 32 32 32 32 32 32 32 00 04 00   DAD2222222222...
0430   41 41 41 36 36 36 36 36 36 36 36 36 36 10 04 01   AAA6666666666...
0440   53 4F 4E 31 31 31 31 31 31 31 31 31 31 00 04 01   SON1111111111...
0450   4D 4F 4D 33 33 33 33 33 33 33 33 33 33 00 04 01   MOM3333333333...
0460   53 49 44 38 38 38 38 38 38 38 38 38 38 40 04 01   SID8888888888@..
0470   41 5A 5A 37 37 37 37 37 37 37 37 37 37 00 04 01   AZZ7777777777...
```

Fig. 9.30: Geketende lijst. Een voorbeeld run.

```
-G220/223
P=0223 0223'            Run "SEARCH" voorgewiste ingang

                        —Niet gevonden
-DR
    N   A=37 BC=00FF DE=0400 HL=0000 S=0100 P=0223 0223' CALL 0171
        A'=00 B'=0000 D'=0000 H'=0000 X=0400 Y=0310 I=00       (0171')
-SY
Y=0310 340   )
-            )
-G220/223    )  Run "SEARCH" voor bestaande ingang
             )
P=0223 0223' )
                  —gevonden
-DR
 Z  N   A=54 BC=FF10 DE=0430 HL=043E S=0100 P=0223 0223' CALL 0171
        A'=00 B'=0000 D'=0000 H'=0000 X=0410 Y=0340 I=00       (0171')
-G226/229
                Delete                   — adres van ingang
P=0229 0229'

                                                           NB: merk de
                                                           veranderingen
-DM400                                                     van pointers
0400  7B 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  {...............
0410  41 4E 54 35 35 35 35 35-35 35 35 35 35 70 04 00  ANT5555555555p..
0420  44 41 44 32 32 32 32 32-32 32 32 32 32 00 04 00  DAD2222222222...
0430  41 41 41 36 36 36 36 36-36 36 36 36 36 70 04 01  AAA6666666666p..
0440  53 4F 4E 31 31 31 31 31-31 31 31 31 31 00 04 01  SON1111111111...
0450  4D 4F 4D 33 33 33 33 33-33 33 33 33 33 00 04 01  MOM3333333333...
0460  53 49 44 38 38 38 38 38-38 38 38 38 38 40 04 01  SID8888888888P..
0470  41 5A 5A 37 37 37 37 37-37 37 37 37 37 00 04 01  AZZ7777777777...
```

Fig. 9.30: Geketende lijst voorbeeld run. (vervolg)

SAMENVATTING

De beginnende programmeur hoeft zich nog niet te bekommeren om de details van data structuren. Het efficient programmeren van niet vanzelf sprekende algoritmen vraagt een zekere kennis van deze structuren. De voorbeelden in dit hoofdstuk zullen de lezer helpen bij het vergaren van deze kennis, waardoor hij (de lezer) alle voorkomende gewone problemen op kan lossen met redelijke data structuren.

10
PROGRAMMA ONTWIKKELING

INLEIDING

Alle programma's die we tot nu toe bestudeerd en ontwikkeld hebben, zijn gemaakt zonder hulp van enige software of hardware. De enige verbetering van de simpele binaire codering is het gebruik van mnemonics geweest van de machine taal. Om efficient software te kunnen ontwikkelen, is het noodzakelijk te weten welke hulpmiddelen (hardware en software) daartoe ter beschikking staan. In dit hoofdstuk zullen we deze hulpmiddelen de revue laten passeren.

BASIS PROGRAMMEER ALTERNATIEVEN

Er zijn drie basis alternatieven waaruit gekozen kan worden: het schrijven van een programma in hex of binaire code, schrijven in een "assembler" taal, of schrijven in een hogere taal.

Hexadecimaal coderen

De meeste programma's zullen geschreven worden in mnemonics van een machine taal. De meeste "single-board" microcomputers zijn echter meestal niet uitgerust met een "assembler" of vertaler. Een assembler is een programma, dat de mnemonics vertaalt in de juiste binaire code. Is er geen assembler, dan moet die vertaling met de hand worden gedaan. Binair is echter *onplezierig* in gebruik, en nodigt uit tot het maken van fouten, zodat meestal hex wordt gebruikt. We hebben in

hoofdstuk 1 gezien, dat een hexadecimaal getal vier binaire bits voorstelt. Voor een byte zijn dus twee hex getallen nodig. Bij wijze van voorbeeld zijn van alle Z80 instructies de hexadecimale codes in de bijlage opgenomen.

Om kort te zijn, als de gebruiker beperkt is in zijn hulpmiddelen, en geen assembler tot zijn beschikking heeft, zal hij het programma met de hand moeten vertalen in hexadecimale code. Dat kan worden gedaan voor een klein aantal instructies, 10 tot 100 misschien. Voor grote programma's is het saai werk, en er worden veel fouten bij gemaakt. Voor de meeste single-board computers zal het toch op deze manier moeten gebeuren, want ze bezitten geen assembler en volledig toetsenbord. Dit om de kosten te beperken.

Samenvattend geldt: hexadecimaal coderen is eigenlijk niet de meest wenselijke manier om een programma in te voeren. Het is vaak alleen economische noodzaak. De kosten van een assembler en een toetsenbord worden afgewogen tegen het extra werk dat gedaan moet worden om het programma hex in het geheugen te zetten. Dit verandert echter niets aan de manier waarop het programma wordt geschreven. *Het programma wordt ook dan geschreven in assembler taal, zodat het voor de menselijke programmeur begrijpelijk blijft.*

Programmeren in assembler taal

Programma's geschreven in assembler taal kunnen zowel met de hand worden vertaald, als in symbolische vorm in de computer gevoerd worden (dit laatste als een assembler aanwezig is). We gaan nu de laatste vorm nader bekijken. De programmeur moet de beschikking hebben over een assembler programma. De assembler leest iedere mnemonic, en vertaalt het in een bit patroon van 1 tot 5 bytes lengte. Een goed assembler programma biedt daarbij nog een aantal andere faciliteiten. We kunnen symbolen gebruiken in plaats van echte getallen, er mogen symbolische adressen worden gebruikt, waar naar gesprongen kan worden. Tijdens het foutzoeken kunnen instructies worden toegevoegd of worden weggehaald, zonder dat het hele programma opnieuw geschreven moet worden. De assembler zorgt dat alles goed gaat. De gebruiker kan het programma in een symbolische vorm foutzoeken. Een disassembler kan gebruikt worden om de inhoud van een geheugen plaats te onderzoeken en de oorspronkelijke instructie weer afleiden uit de code. Al deze software hulpmiddelen zullen we later bestuderen. Nu kijken we eerst eens naar het derde alternatief.

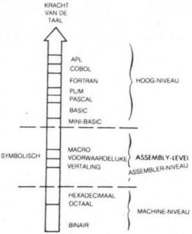


Fig. 10.1: Niveaus in programmeren

Hogere taal

Een programma kan geschreven zijn in een hogere taal, zoals BASIC, APL, PASCAL, en nog vele andere. Er zijn boeken geschreven over deze talen , en we gaan in dit boek er ook niet dieper op in. We willen ons hier beperken tot een kort overzicht. Een hogere taal heeft vele krachtige instructies, die het programmeren sneller en gemakkelijker maken. Deze instructies worden door een ingewikkeld programma vertaald in de binaire code, die de microprocessor begrijpt. Iedere instructie in de hogere taal wordt meestal vertaald in een groot aantal binaire instructies. Een programma dat deze automatische vertaling uitvoert, is een *compiler* of een *interpreter*. Een compiler vertaalt de instructies van de hogere taal in machine instructies, de object code. Zijn alle instructies vertaald, dan wordt in een aparte fase het programma uitgevoerd. Een interpreter daarentegen, vertaalt een instructie, voert deze uit, haalt de volgende instructie, vertaalt deze, en voert hem uit, enz. Een interpreter werkt interactief, wat een voordeel kan zijn. Het nadeel is de lage efficientie van de gegenereerde code, dat in tegenstelling tot de compiler. We zuller daar op deze plaats niet verder op in gaan. We keren nu terug naar het programmeren van de microprocessor op het niveau van de assembler taal.

SOFTWARE ONDERSTEUNING

In dit hoofdstuk komen de belangrijkste software faciliteiten aan de orde, die in een compleet systeem aanwezig zijn (of aanwezig zouden moeten zijn), om op een gemakkelijke manier programma's te ontwikkelen. Enkele definities zijn al gegeven. We zullen ze nog eens herhalen, en we zullen er enkele aan toegevoegen.

De *assembler* is het programma, dat de mnemonics vertaalt in hun binaire code. Een symbolische instructie wordt meestal vertaald in een binaire instructie (die 1, 2 of 3 bytes lang kan zijn). De binaire code heet de *object code*. Deze code kan door de microcomputer worden uitgevoerd. Een neven effect is, dat de assembler een complete symbolische listing produceert van het programma, een equivalentie tabel, en een lijst met de plaatsen waar de symbolen in het programma voorkomen. Later in dit hoofdstuk zullen we daar voorbeelden van tegenkomen.

De assembler geeft bovendien een lijst met syntax fouten, zoals instructies die fout zijn gespeld, of helemaal niet bestaan, sprong fouten, dubbele labels of niet bestaande labels.

De assembler ziet geen *logische* fouten (dat is je eigen probleem).

Een *compiler* is een programma, dat hogere taal instructies vertaalt in hun binaire vorm.

Een *interpreter* lijkt op de compiler. Deze vertaalt ook een hogere taal in binaire instructies. In tegenstelling tot de compiler voert de interpreter iedere vertaalde instructie meteen uit. Daarbij kan een tussen code gebruikt worden, maar vaak zal dat niet voorkomen.

Een *monitor* is een basis programma, dat onmisbaar is voor het gebruiken van de hardware van het systeem. Het bestuurt de I/O van het systeem. Bijvoorbeeld, een minimale monitor van een single-board computer met een hex toetsenbord en LED's controleert voortdurend het toetsenbord en geeft de data afkomstig van dat toetsenbord weer op de LED's. Bovendien moet de monitor enkele commando's afkomstig van het toetsenbord begrijpen en kunnen uitvoeren, zoals START, STOP, DOORGAAN, LAAD GEHEUGEN, en ONDERZOEK GEHEUGEN. Een monitor in grote systemen heeft veel meer taken, en wordt dan het "*operating system*" genoemd. Operating systemen die werken met een schijf, of disk, heten *disk operating system*, afgekort DOS.

Een *editor* zorgt voor gemakkelijke invoer en wijziging van tekst of programma's. We kunnen ermee karakters toevoegen, wissen, lijnen toevoegen en wissen, reeksen karakters zoeken en vervangen, enz.

Een goede editor is een belangrijk hulpmiddel bij het invoeren van programma's.

Een foutzoeker (debugger) helpt het programma foutloos te maken. Als een programma niet juist werkt, is er vaak niet direct een oorzaak voor te vinden. De programmeur zal willen, dat het programma op bepaalde adressen stopt (breakpoints), zodat hij de registers of het geheugen op dat punt kan controleren. Dat is primair de functie van de foutzoeker. Een goede foutzoeker kan meer, zoals de keuze laten in welke vorm de programmeur de data wil zien: symbolisch, hex of binair, of nog andere voorstellingen.

Een *loader* of *linking loader* kan verschillende blokken object code op gespecificeerde adressen in het geheugen plaatsen, en de verschillende pointers veranderen, zodat de blokken elkaar kunnen refereren.

Een *simulator* of *emulator* programma wordt gebruikt om de werking van een chip te simuleren, meestal de microprocessor. Het te testen programma loopt dan op een gesimuleerde processor. Op deze wijze wordt het mogelijk het programma tijdelijk te stoppen, en te veranderen. De nadelen van de simulator zijn:

1 - simuleert gewoonlijk alleen de microprocessor, en geen input/output chips.

2 - werkt meestal traag, en ook de tijd wordt gesimuleerd.

Er kunnen daarom geen "real-time" chips mee worden getest, en er kunnen synchronisatie problemen blijven bestaan, hoewel de logica van het programma correct kan zijn.

Een *emulator* is in essentie een simulator in "echte" tijd. Er wordt een processor gebruikt om een andere te simuleren, tot in alle details.

Gebruiks routines zijn eigelijk alle routines die nodig zijn in de toepassingen en waarvan de gebruiker wenst, dat de fabrikant ze had geleverd!

Deze routines kunnen zijn: vermenigvuldigen, delen, en andere rekenkundige bewerkingen, blok verplaatsingen, testen van karakters, input/output routines (of "drivers"), en nog andere.

HET ONTWIKKELEN VAN EEN PROGRAMMA

We gaan nu eens kijken naar een typische volgorde, waarin een programma op assembler taal niveau ontwikkeld wordt. We nemen aan, dat alle gebruikelijke software hulpmiddelen aanwezig zijn, om hun nut te

kunnen demonstreren. Het is weliswaar mogelijk programma's te maken zonder deze hulpmiddelen, maar het gemak waarmee dat gaat zal duidelijk minder zijn, en er zal meer tijd in het foutzoeken gestoken moeten worden.

De eerste benadering is het ontwerp van een algoritme en een data structuur voor het op te lossen probleem. De volgende stap is een begrijpelijk stroomdiagramma. Tenslotte moet het stroomdiagramma vertaald worden in het programma. Dat wordt de codeer fase genoemd.

Daarna moet het programma in de computer gevoerd worden. De hardware, die daarvoor nodig is, komt in het volgende deel van dit hoofdstuk aan bod.

Het programma komt in RAM te staan, onder controle van de editor. Ieder deel van het programma, subroutines bijvoorbeeld, kunnen dan worden getest.

Allereerst wordt de assembler gebruikt. Was deze nog niet aanwezig in het systeem, dan moet deze eerst geladen worden van een extern geheugen, bijvoorbeeld een schijf. Daarna wordt het programma vertaald in een binaire code. De aldus ontstane object code kan worden uitgevoerd door de computer.

Meestal mag je niet verwachten, dat een programma meteen goed is. Om de juiste werking te testen, worden "breakpoints" gezet op belangrijke adressen van het programma. Op deze wijze kunnen tussentijdse resultaten getest worden. De foutzoeker (debugger) wordt voor dit doel gebruikt.

Het programma begint na het "Go" commando. Bij ieder breakpoint zal het stoppen. De programmeur kan dan registers en geheugen controleren, of ze de juiste inhoud hebben. Is dat het geval, dan gaan we door tot het volgende breakpoint. Vinden we data die niet klopt, dan zit er een fout in het programma. De programmeur zal dan eerst kijken of hij de goede instructies heeft gebruikt. Kan hij daarbij geen fout vinden, dan is er misschien sprake van een logische fout. Het stroomdiagram moet dus worden gecontroleerd. Laten we hier aannemen, dat het stroomdiagram juist is. Er is dus iets fout gegaan bij het coderen. Het zal nodig zijn een deel van het programma te veranderen. Zit de symbolische voorstelling van het programma nog in het geheugen, dan kunnen we heel eenvoudig de nodige regels m.b.v. de editor veranderen, en de zojuist beschreven handelingen herhalen. In sommige systemen echter, is niet voldoende geheugen aanwezig, zodat de symbolische voorstelling eerst op schijf of cassette gezet moet worden, voordat de object code uitgevoerd kan worden. In dat geval moet natuurlijk

het programma opnieuw geladen worden, voordat we met de editor veranderingen aan kunnen brengen.

De bovenstaande procedure moet herhaald worden, net zolang tot het programma goed is. Maar ook hier geldt: voorkomen is beter dan genezen. Een goed ontwerp resulteert meestal in een snel foutloos programma. Een gammel ontwerp als uitgangspunt betekent echter een ware lijdensweg. De tijd, die je nodig hebt om het programma foutloos te maken, zal vele malen groter zijn dan de tijd die je nodig had voor het ontwerp. Je kunt dus beter iets meer tijd besteden aan het ontwerp van je programma, om het foutzoeken korter te maken.

Met deze benadering kan heel goed de algemene opbouw van het programma worden getest, maar het testen van het programma met de I/O apparatuur in real time is niet mogelijk. Een oplossing is, het programma in EPROM zetten, en gewoon kijken of het werkt.

Maar er is een betere oplossing: de *in-circuit emulator*. Zo'n emulator gebruikt een Z80 (of een andere processor) om de Z80 in de computer te simuleren in real time. De Z80 wordt uit de computer gehaald, en daarvoor in de plaats komt een 40 pins connector van de emulator. De signalen afkomstig van de emulator komen volledig overeen met die van de Z80, alleen misschien iets langzamer. Het belangrijke voordeel hiervan is, dat het programma in RAM blijft. Nu kan het programma wel getest worden in relatie met de I/O apparatuur. Bovendien staan de programmeur alle hulpmiddelen ter beschikking, zoals de editor, foutzoeker, symbolische faciliteiten, enz.

Een goede emulator heeft meer faciliteiten, zoals een *trace*. Een trace is een opname van de laatste instructies, of de status van de verschillende bussen in het systeem vlak voor een breakpoint. Het kan zelfs een scoop triggeren op een gespecificeerd adres, of een bepaalde combinatie van bits. Zo'n faciliteit heeft een grote waarde, want als de fout zichtbaar wordt, is het meestal al te laat. De oorzaak van de fout, een instructie of data, komt al eerder in het programma voor. M.b.v de trace is het mogelijk uit te vinden in welk deel van het programma de oorzaak van de fout zit. Is de trace niet lang genoeg, dan kan het breakpoint eenvoudig eerder gezet worden.

De ontwikkeling van een programma is nu beschreven. Als volgende, gaan we de hardware alternatieven voor het ontwikkelen van programma's nader bekijken.

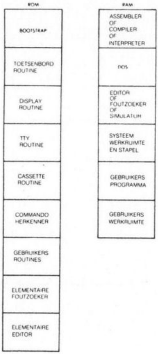


Fig. 10.2: Een typische "plattegrond" van het geheugen

HARDWARE ALTERNATIEVEN

Single-board microcomputer

Een single-board computer (een microcomputer op 1 kaart) is de voordeligste benadering. Er is meestal een hexadecimaal toetsenbord en 6 LED's, waarmee adres en data zichtbaar worden gemaakt, op zo'n single-board computer. Vanwege de kleine hoeveelheid geheugen, is er meestal geen assembler aanwezig. Hoogstens heeft het een kleine monitor, en vrijwel geen edit en foutzoek faciliteiten. Alle programma's moeten in hexadecimale vorm ingevoerd worden. In principe kan een single-board computer hetzelfde als zijn grote broers. Maar van-

wege de kleine hoeveelheid aanwezig geheugen biedt het niet de zelfde faciliteiten, en programma ontwikkeling duurt dan ook veel langer. Daarom is een dergelijke computer het best geschikt voor het onderwijs, en daar waar slechts kleine programma's gemaakt moeten worden. Single-board computers zijn waarschijnlijk de goedkoopste manier om het programmeren te leren. Ze zijn echter niet geschikt voor het ontwikkelen van grote programma's, tenzij ze uitgebreid zijn met meer geheugen en de benodigde ontwikkel faciliteiten.

Het ontwikkel systeem

Een ontwikkel systeem is een microcomputer met een grote hoeveelheid geheugen (32K, 48K), een aantal randapparaten, zoals een beeldscherm (CRT), printer, schijven en een PROM programmer, en misschien een in-circuit emulator. Een ontwikkel systeem is speciaal bedoeld voor de ontwikkeling van programma's in een industriele omgeving. Het heeft alle, of bijna alle, software faciteiten, zoals we die besproken hebben in het vorige deel. In principe is het een ideaal instrument voor de ontwikkeling van software.

Het ontwikkel systeem is misschien in zoverre beperkt, dat het geen compiler of interpreter heeft, omdat daarvoor veel geheugen nodig is, meer dan het systeem bezit. Het systeem biedt echter alle faciliteiten voor het ontwikkelen van programma's in assembler taal. Omdat deze ontwikkel systemen veel minder verkocht worden dan hobby computers, zijn ze ook veel duurder.

Hobby computers

De hardware van een hobby computer is natuurlijk analoog aan die van een ontwikkel systeem. Het verschil zit in de software. Deze is veel minder geavanceerd dan die van een ontwikkel systeem. Hobby computers zijn daarom te zien als een tussenstap tussen de single-board computer en het microcomputer ontwikkel systeem. Voor mensen, die bescheiden programma's willen maken, zijn ze de beste keus. Deze computers zijn vrij "goedkoop" en bieden toch een aantal bruikbare software hulpmiddelen.

Time-sharing systeem

Het is mogelijk bij bepaalde maatschappijen terminals te huren, die opgenomen zijn in een time-sharing computer netwerk. Alle terminals

delen de beschikbare tijd van een grote computer, en hebben alle voordelen van een dergelijke grote computer. De meeste systemen hebben *cross assemblers* voor vrijwel alle microprocessors. Een cross assembler is een assembler voor, bijvoorbeeld, de Z80 op, bijvoorbeeld, een IBM370. Of formeel gezegd, een cross assembler is een assembler voor microprocessor X, op processor Y. Welke processor Y gebruikt wordt is niet van belang. De gebruiker schrijft gewoon assembler taal programma's voor de Z80, en de cross assembler vertaalt het in het goede binaire patroon. Het programma kan op dat punt echter niet uitgevoerd worden, tenzij er een programma is, dat microprocessor X simuleert. I/O apparatuur kan dan niet worden getest. Deze oplossing wordt daarom alleen in industriele omgevingen gebruikt.

De bedrijfs computer

Heeft een bedrijf een eigen grote computer, dan zijn meestal wel cross assemblers aanwezig. Is er sprake van time sharing, dan komt dit overeen met de hierboven besproken situatie. Wordt er in batch gewerkt, dan is dat misschien wel de meest ongeschikte manier om programma's te ontwikkelen, omdat daarmee erg veel tijd gemoeid is.

Wel of geen bedieningspaneel?

Een bedieningspaneel is een stuk hardware, dat vaak voor het foutzoeken gebruikt wordt. Oorspronkelijk was het een hulpmiddel voor het weergeven van de binaire inhoud van een register of het geheugen. Alle functies van een bedieningspaneel echter, kunnen worden uitgevoerd door een terminal, zodat deze tegenwoordig vaak wordt gebruikt. Het voordeel van een CRT is, dat gekozen kan worden tussen de verschillende soorten voorstellingen, zoals binair, hex, symbolisch en decimaal, mits de routines daarvoor aanwezig zijn. Het nadeel is, dat meerdere toetsen aangeslagen moeten worden, in plaats van 1 knop. Omdat de kosten echter van een bedieningspaneel nog wel aanzienlijk zijn, wordt het vrijwel niet meer toegepast als hulpmiddel bij het foutzoeken. De waarde die men vaak hecht aan een bedieningspaneel is vaak op basis van emotionele argumenten in plaats van op verstandelijke. Ik vind, dat een bedieningspaneel niet onmisbaar is.

Samenvatting van de hardware hulpmiddelen

Er zijn drie gevallen te onderscheiden. Heb je slechts een beperkt

budget, en wil je leren programmeren, koop dan een single-board computer. Je zult alle programma's in dit boek ermee kunnen maken, en nog vele andere. Wil je echter programma's maken met meer dan een paar honderd instructies, dan zul je zeker last hebben van de beperkingen van een dergelijk systeem.

Ben je een industrieel gebruiker, dan heb je een ontwikkel systeem nodig. Iedere besparing op een dergelijk systeem betekent een aanzienlijk langere ontwikkel tijd. De keuze is duidelijk: hardware of ontwikkeltijd. Natuurlijk, als de programma's eenvoudig zijn, kan een goedkopere oplossing worden gezocht. Als er echter grote ingewikkelde programma's gemaakt moeten worden, dan is een besparing op het ontwikkel systeem nauwelijks te rechtvaardigen. Programmeer kosten maken het grootste deel van alle kosten uit.

Voor de amateur is de hobby computer, met zijn minimale, maar voldoende faciliteiten, een goede keus. Voor veel hobby computers is al goede ontwikkel software te krijgen.

Maar nu gaan we het meest onmisbare hulpmiddel bekijken: de assembler.

DE ASSEMBLER

We hebben het hele boek door assembler taal gebruikt, zonder de formele syntax of definitie ervan te geven. Dat verzuim zal hier goed worden gemaakt. Een assembler is ontworpen om symbolische voorstelling van een programma mogelijk te maken. Toch moet het assembler programma de mnemonics gemakkelijk kunnen vertalen in de binaire voorstelling.

Assembler velden

We hebben gezien, dat verschillende velden worden gebruikt:

Het *label veld*, facultatief, kan een symbolisch adres bevatten voor de erop volgende instructie.

Het *instructie veld*, met de opcode en operanden (soms hebben de operanden een eigen veld).

Het *commentaar veld*, is het meest rechtse veld, facultatief, en bedoeld voor de documentatie van het programma.

Deze velden worden getoond in het programmeer formulier in figuur 10.3.

ADRES	HEX INSTRUCTIE				SYMBOLISCH			COMMENTAAR
	1	2	3	4	LABEL	OPCODE	OPERAND	

Fig. 10.3: Microprocessor programmeer formulier

Zit het programma eenmaal in de assembler, dan produceert deze er een *listing* van. De assembler voegt aan de genoemde velden drie velden toe, meestal links op de pagina. Zie de listing in figuur 10.4. Links staat het regel nummer. Iedere door de programmeur ingetypte regel krijgt een symbolisch regel nummer.

Het volgende veld geeft het echte adres, d.w.z. de hexadecimale waarde van de programma teller als deze naar de instructie wijst.

Nog verder naar rechts vinden we de hexadecimale voorstelling van de instructie.

Zelfs al maak je programma's voor een single-board computer met hex invoer, dan nog kun je het programma schrijven in assembler taal, mits je toegang hebt tot een systeem met een assembler. Daarna kun je het programma uitvoeren op je eigen systeem, want de assembler genereert de juiste hexadecimale codes voor je systeem. Dit eenvoudige voorbeeld toont het nut van een dergelijk software hulpmiddel.

Tabellen

Wanneer de assembler het programma vertaalt, voert het twee belangrijke taken uit:

1 - het vertaalt de mnemonics in de binaire code.
2 - het vertaalt de symbolen voor constantes en adressen in hun echte binaire waarden.

De assembler geeft i.v.m. het foutzoeken aan het eind van de listing een tabel met daarin de symbolen en hun hexadecimale waarde. Deze tabel heet de "symbol table".

Sommige symbol tables geven niet alleen symbool en waarde, maar ook de regels waar ze in voorkomen.

Fout boodschappen

Tijdens het vertalen ontdekt de assembler syntax fouten en geeft deze als onderdeel van de listing. Foutboodschappen kunnen zijn: niet gedefinieerd symbool, label al gedefinieerd, foute opcode, fout adres, foute adresserings mode. Meer gedetailleerde foutmeldingen zijn natuurlijk wenselijk, en zijn dan ook vaak voorhanden. Ze varieren echter van assembler tot assembler.

De assembler taal

De opcodes zijn al eerder gedefinieerd. We zullen hier de symbolen, constantes en operatoren definieren, die deel uit kunnen maken van de syntax van de assembler.

Symbolen

Symbolen worden gebruikt om numerieke waarden (data of adressen) voor te stellen. Ze mogen bestaan uit ten hoogste 6 karakters, en moeten met een letter beginnen. De karakters mogen zijn: de letters uit het alfabet en getallen. Namen identiek aan opcodes mogen niet gebruikt worden. A, B, C, D, E, H, L, BC, DE, HL, AF, IX, IY, SP en de korte namen, die als pseudo operators door de assembler worden gebruikt, zijn niet toegestaan. De afkortingen van de verschillende vlagen mogen ook niet als symbolen gebruikt worden: C, Z, N, PE, NC, P, PO.

Het toekennen van een waarde aan een symbool

Labels zijn speciale symbolen, waarvan de waarde niet gedefinieerd hoeft te worden door de programmeur. De assembler doet dat automatisch, zogauw deze een label tegenkomt. De label krijgt op deze wijze de waarde van de regel waarin deze staat. Er bestaan speciale pseudo-instructies, waarmee nieuwe start waarden voor de labels kunnen worden geforceerd, of waarmee ze een specifieke waarde krijgen.

```
CROMEMCO CDOS Z80 ASSEMBLER version 02.15                    PAGE 0001

0000'              0001           ORG    0100H
      (0200)       0002 MPRAD     DL     0200H
      (0202)       0003 MPDAD     DL     0202H
      (0204)       0004 RESAD     DL     0204H
                   0005 ;
0100  ED4B0002     0006 MP488     LD     HL,(MPRAD)   ;LOAD MULTIPLIER INTO C
0104  0608         0007           LD     B,8          ;B IS BIT COUNTER
0106  ED5B0202     0008           LD     DE,(MPDAD)   ;LOAD MULTIPLICAND INTO E
010A  1600         0009           LD     D,0          ;CLEAR D
010C  210000       0010           LD     HL,0         ;SET RESULT TO 0
010F  CB39         0011 MULT      SRL    C            ;SHIFT MULTIPLIER BIT INTO CARRY
0111  3001         0012           JR     NC,NOADD     ;TEST CARRY
0113  19           0013           ADD    HL,DE        ;ADD MPD TO RESULT
0114  CB23         0014 NOADD     SLA    E            ;SHIFT MPD LEFT
0116  CB12         0015           RL     D            ;SAVE BIT IN D
0118  05           0016           DEC    B            ;DECREMENT SHIFT COUNTER
0119  C20F01       0017           JP     NZ,MULT      ;DO IT AGAIN IF COUNTER <> 0
011C  220402       0018           LD     (RESAD),HL   ;STORE RESULT
011F  (0000)       0019           END

Errors             0
```

Fig. 10.4: Een voorbeeld van een assembler listing

Symbolen voor constanten en geheugen adressen moeten echter door de programmeur worden gedefinieerd, voordat ze gebruikt worden.

Voor het toekennen van waarden aan symbolen kan een *directive* worden gebruikt. (Directive betekent richtlijn). Een directive is een aanwijzing voor de assembler, die niet wordt vertaald in een uitvoerbare instructie code. We willen, bijvoorbeeld, het symbool LOG een waarde toekennen. Dat kan als volgt:

```
LOG  DFW  3002H
```

LOG krijgt de waarde 3002 hexadecimaal. De assembler directives worden later in dit hoofdstuk onderzocht.

Constanten of literals

Constanten kunnen worden uitgedrukt in decimale, hexadecimale, octale of in binaire voorstelling, en zelfs als een reeks alfanumerieke karakters. Daarom moet aan het getal een symbool worden toegevoegd, om aan te geven welke basis gebruikt wordt. Om 0 in de accumulator te laden kunnen we eenvoudig schrijven:

```
LD  A, 0
```

eventueel gevolgd door een D.

Een hexadecimaal getal moet eindigen met een H. Om FF in de accumulator te laden moeten we schrijven:

```
LD  A, FFH
```

Een octaal getal eindigt met O of Q, een binair getal met B. Stel we willen bijvoorbeeld 11111111 in de accumulator laden:

```
LD  A,  11111111B
```

In het literal veld mogen ook ASCII karakters worden gebruikt. Dit karakter moet tussen apostrofs geplaatst worden. Bijvoorbeeld:

```
LD  A,  'S'
```

Oefening 10.1: Laden de volgende twee instructies de zelfde waarde in de accumulator? LD A,'5' en LD A,5H.

Denk eraan, dat volgens Zilog afspraken, ronde haken een adres weergeven. Bijvoorbeeld:

```
LD  A,  (10)
```

geeft aan, dat de inhoud van adres 10 in de accumulator geladen moet worden.

Operatoren

Om het maken van symbolische programma's nog verder te vergemakkelijken, staat de assembler het gebruik van zogenaamde operatoren toe. Tenminste moeten de plus en de min toegestaan worden. Dan is het volgende mogelijk:

```
LD  A, (ADDRESS)
LD  A, (ADDRESS +1)
```

De uitdrukking ADRES + 1 wordt door de assembler uitgerekend, om het echte adres (binair) te kunnen vinden. Het wordt in *assembler tijd* uitgerekend, en niet tijdens de uitvoering van het programma.

Andere operatoren kunnen aanwezig zijn, zoals vermenigvuldigen en delen, wat erg gemakkelijk is bij tabellen. Ook komen soms voor groter dan, en kleiner dan.

De uitdrukking moet, natuurlijk, altijd resulteren in een positief antwoord. Negatieve uitkomsten mogen normaal niet worden gebruikt. Deze moeten uitgedrukt worden in het hexadecimale formaat.

Tenslotte is er een symbool, dat traditioneel gebruikt wordt om de huidige waarde van de programma teller uit te drukken: "$".

Oefening 10.2: Wat is het verschil tussen de volgende instructies?

```
LD  A, 10101010B
LD  A, (10101010B)
```

Oefening 10.3: Wat is het effect van de instructie: JP NC,$ – 2

```
JP  NC, $ - 2
```

Uitdrukkingen

De Z80 assembler laat een groot aantal uitdrukkingen met rekenkundige en logische bewerkingen toe. De assembler werkt deze uitdrukkingen uit van links naar rechts, met in acht neming van de prioriteiten zoals deze in figuur 10.5 voorkomen. Ronde haken mogen voorkomen, om een bepaalde prioriteit af te dwingen, maar de buitenste haken geven altijd aan, dat het om een adres gaat.

Assembler directives

Directives zijn opdrachten van de programmeur aan de assembler, waardoor, bijvoorbeeld, waarden aan symbolen toegekend worden, of waarmee print opdrachten gegeven kunnen worden. Deze laatste opdrachten worden commando's genoemd, en zullen apart worden besproken.

We zullen nu de 11 assembler directives van het Z80 ontwikkel systeem de revue laten passeren:

ORG nn

De assembler adres teller wordt op de waarde nn gezet. Dat wil zeggen, dat de eerstvolgende instructie na deze directive op adres nn komt. Het kan gebruikt worden, om verschillende delen van het programma op verschillende adressen in het geheugen te plaatsen.

EQU nn

Wordt gebruikt om een label een waarde te geven.

DEFL nn

Deze directive geeft ook een label een waarde, maar mag meerdere keren in een programma voorkomen, om het zelfde label verschillende waarden te geven, terwijl EQU maar een maal mag voorkomen per label.

DEFB n

De waarde n wordt toegekend aan de inhoud van het adres waar deze directive staat.

DEFB 'S'

Kent de ASCII waarde toe aan het byte.

DEFW nn

Kent de waarde nn toe aan het twee bytes woord op het huidige adres.

DEFS nn

Reserveert in het geheugen een blok van nn bytes, te beginnen op het huidige adres.

DEFM 'S'

De reeks karakters tussen de apostrofs wordt in het geheugen ge-

OPERATOR	FUNCTIE	PRIORITEIT
+	PLUS	1
−	MINUS	1
.NOT. or \	LOGISCHE NEGATIE	1
.RES.	RESULTAAT	1
**	EXPONENT	2
*	VERMENIGVULDIGING	3
/	DELING	3
.MOD.	MODULO	3
.SHR.	LOGISCHE VERSCHUIVING RECHTS	3
.SHL.	LOGISCHE VERSCHUIVING LINKS	3
+	OPTELLING	4
-	AFTREKKING	4
.AND. or &	LOGISCHE EN	5
.OR. or !	LOGISCHE OF	6
.XOR.	LOGISCHE EXCLUSIEVE OF	6
.EQ. or =	GELIJK	7
.GT. or >	GROTER DAN	7
.LT. or <	KLEINER DAN	7
.UGT.	GROTER DAN ZONDER TEKEN	7
.ULT.	KLEINER DAN ZONDER TEKEN	7

Fig. 10.5: Prioriteiten

plaatst, te beginnen op het huidige adres. De reeks moet kleiner zijn dan 63 karakters.

MACRO P0 P1 ..... Pn

Wordt gebruikt om een label als macro te definieren, tesamen met zijn parameters. Dit komt later nog ter sprake.

END

Geeft het einde van een programma aan. Instructies na deze directive worden genegeerd.

ENDM

Geeft het einde van een macro definitie aan.

Assembler commando's

Commando's worden gebruikt om het formaat van de listing te veranderen. Alle commando's beginnen met een ster in kolom 1. De Z80 assembler kent zeven commando's. Voorbeelden hiervan zijn:

EJECT

De listing gaat verder aan het begin van de volgende pagina.

LIST OFF

Het printen van de listing houdt op bij dit commando.

De andere commando's zijn: "*HEADING S", "*LIST ON", "*MACLIST ON", "*MACLIST OFF", "*INCLUDE FILENAME".

Macro's

Een macro is eenvoudig gezegd, een naam toegekend aan een groep instructies. Dit is gemakkelijk voor de programmeur. Wordt een bepaalde groep instructies herhaald gebruikt in een programma, dan kan een macro worden gedefinieerd, waardoor niet iedere keer alle instructies geschreven hoeven te worden. Een voorbeeld van een macro is:

```
SAVREG MACRO PUSH AF
             PUSH BC
             PUSH DE
             PUSH HL
             ENDM
```

Iedere keer als we in het programma de registers op de stapel willen zetten, hoeven we alleen maar te schrijven: "SAVREG" i.p.v. alle instructies. Iedere keer als de assembler de naam SAVREG tegenkomt, zal deze de naam vervangen door de vier instructies. Een assembler met een dergelijke faciliteit heet een macro-assembler.

Macro of subroutine

De macro lijkt in werking misschien op een subroutine, maar dat is niet het geval. Als de assembler de object code genereert, vervangt deze alle macro's door de daadwerkelijke instructies. Tijdens het uit-

voeren van het programma verschijnt iedere groep instructies zo vaak als de macro voorkomt.

Een subroutine, daarentegen, komt slechts een maal in het programma voor. Hij wordt alleen herhaald gebruikt. Het programma springt daarvoor naar het adres van de subroutine. Een macro wordt tijdens het vertalen geexpandeerd, een subroutine wordt uitgevoerd tijdens de executie van het programma. Ze werken op een verschillende manier.

Macro parameters

Iedere macro kan voorzien worden van een aantal parameters. Laten wij, bij wijze van voorbeeld, de volgende macro bekijken:

```
SWAP  MACRO   #M, #N, #T
      LD      A, #M        M INTO A
      LD      #T, A        A INTO T (=M)
      LD      A, #N        N INTO A
      LD      #M, A        A INTO M (=N)
      LD      A, #T        T INTO A
      LD      #N, A        A INTO N (=T)
      END     M
```

Deze macro verwisselt de inhouden van de geheugen adressen M en N. De Z80 is niet voorzien van een dergelijke instructie. Een macro is dan een oplossing. "T" wordt hier gebruikt als een tijdelijke opslag. We willen nu de inhouden van de adressen ALPHA en BETA verwisselen. De instructie die we dan moeten gebruiken is:

```
SWAP (ALPHA), (BETA), (TEMP)
```

TEMP is het adres voor de tijdelijke opslag. De assembler expandeert deze macro tot de volgende instructies:

```
LD  A, (ALPHA)
LD  (TEMP), A
LD  A, (BETA)
LD  (ALPHA), A
LD  A, (TEMP)
LD  (BETA), A
```

Het nut van de macro moet nu duidelijk zijn. De programmeur kan zijn eigen pseudo-instructies gebruiken, die hij m.b.v. macro's definieert. Hij kan als het ware de instructie set van de Z80 uitbreiden, zoveel als hij zelf wil. Er is echter een nadeel. Iedere macro wordt geexpandeerd

tot het aantal instructies dat de programmeur er voor gebruikt heeft. Een macro is dus langzamer dan een enkele instructie. Voor lange programma's zijn macro's echter erg gemakkelijk.

Aanvullende macro faciliteiten

Vele aanvullende macro faciliteiten kunnen aanwezig zijn. Macro's kunnen, bijvoorbeeld, genest voorkomen, d.w.z. een macro definitie binnen een andere. Daardoor kan een macro zichzelf veranderen! De eerste macro aanroep is dan de normale expansie, maar iedere volgende aanroep krijgt een andere expansie van de zelfde macro. Dit mag bij de Z80 assembler, maar geneste definities zijn niet toegestaan.

VOORWAARDELIJKE VERTALING

Een andere faciliteit van de Z80 assembler is de voorwaardelijke vertaling. De programmeur kan zijn programma's ontwerpen met het oog op vele toepassingen. Voor een bepaalde toepassing kan hij dan aangeven, welke onderdelen van het programma vertaald moeten worden. Bijvoorbeeld, een industriele gebruiker ontwerpt programma's voor de besturing van een willekeurig aantal verkeerslichten per kruising. De ontwerper ontvangt de specificaties van een bepaald kruispunt m.b.t. de eisen die aan de verkeerslichten gesteld worden. Door middel van een voorwaardelijke vertaling, worden dan alleen die delen van het programma vertaald, die nodig zijn voor die situatie. Het resultaat is een ideaal passende oplossing voor het probleem.

Voorwaardelijke vertaling is daarom bijzonder nuttig voor het maken van programma's in een industriele omgeving, waar veel variaties mogelijk zijn op een basis ontwerp. Het verzekert een snel antwoord (in de vorm van een programma) op veranderingen van externe parameters.

De micro-assembler van Zilog kent twee voorwaardelijk pseudo-OP's:

COND nn en ENDC

nn geeft de voorwaarde of conditie aan. Zolang de conditie waar is na COND nn, worden de instructies die er op volgen vertaald. Is de conditie echter niet waar, dan worden de instructies tot de volgende ENDC niet vertaald. ENDC wordt gebruikt, om een COND te beeindigen. COND mag niet genest worden.

In theorie kunnen andere voorwaardelijke vertaal faciliteiten voorkomen, met een "IF" en "ELSE" specificatie. Misschien zijn deze beschikbaar in een toekomstige versie van de assembler.

SAMENVATTING

In dit hoofdstuk hebben we kennis gemaakt met de technieken en hulpmiddelen, die nodig zijn bij het ontwikkelen van programma's, tesamen met de beperkingen en alternatieven.

De hardware kan daarbij varieren van een single-board computer tot een volwaardig ontwikkel systeem. Het programmeren kan gebeuren van binair niveau tot op het niveau van een hogere taal.

Je zult hier zelf uit moeten kiezen, op basis van het doel dat je voor ogen staat, en de hulpmiddelen waarover je beschikt.

11
CONCLUSIE

We hebben ons nu bezig gehouden met alle belangrijke aspecten van het programmeren, van definities en basis beginselen tot de interne manipulatie van de Z80 registers, de controle van I/O apparatuur, en de eigenschappen van de hulpmiddelen voor software ontwikkeling. Wat is de volgende stap? Ik kan daar twee gezichtspunten bieden. De eerste heeft betrekking op de ontwikkeling van de technologie, de tweede heeft betrekking op de ontwikkeling van je eigen kennis en vakmanschap.

TECHNOLOGISCHE ONTWIKKELING

De steeds voortschrijdende MOS technologie maakt het mogelijk steeds meer ingewikkelde chips te maken. De daarbij optredende kosten worden steeds lager. Het gevolg is, dat vele input/output chips een ingebouwde eenvoudige processor hebben. Dat betekent, dat de meeste LSI chips in een systeem *programmeerbaar* zullen worden. We staan nu voor een schijnbaar, maar wel interessant dilemma. Om het ontwerpen van software eenvoudiger te maken, en om het aantal componenten te beperken, moeten de nieuwe I/O chips wel programmeerbaar zijn. Daar staat tegenover, dat al deze chips verschillend zijn. Iedere chip moet anders geprogrammeerd worden. De programmeur moet al deze chips dus tot in details bestuderen! *Het programmeren van een systeem betekent niet alleen het programmeren van de microprocessor, maar ook het programmeren van alle andere chips in het systeem.* De tijd, die nodig is voor het leren programmeren van iedere chip, kan aanzienlijk zijn.

Natuurlijk is dit slechts een schijnbaar dilemma. Zouden deze chips niet bestaan, dan zouden de te bouwen interfaces nog ingewikkelder zijn. En dat niet alleen, ook de programma's zouden complexer worden. De complexiteit van de nu ontstane situatie is het gevolg van de veelheid aan de te programmeren chips. Je moet niet 1, maar veel processors programmeren in het systeem. Ik hoop echter, dat met de hulp van dit boek die taak aanzienlijk lichter is geworden.

DE VOLGENDE STAP

"Op papier" ken je nu alle basis technieken die nodig zijn om programma's te ontwikkelen voor eenvoudige toepassingen. Dat was ook het doel van dit boek. De volgende stap is, dat je daadwerkelijk gaat programmeren. Daarvoor is nog geen vervanging uitgevonden, het is onmisbaar. Het is onmogelijk het programmeren alleen uit een boek te leren, je moet ook ervaring opdoen. Maar je bent nu tenminste in staat je eigen programma's te schrijven. Ik hoop, dat je er erg veel plezier aan zult beleven.

Voor degenen, die meer willen lezen over de Z80, wordt een boek beschikbaar in 1983, "The Z80 Applications Book", met daarin vele toepassingen die uitgevoerd kunnen worden op een echte microprocessor.

BIJLAGE A

HEXADECIMALE CONVERSIE TABEL

HEX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	00	000
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	0
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	256	4096
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	512	8192
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	768	12288
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	1024	16384
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	1280	20480
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	1536	24576
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	1792	28672
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	2048	32768
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	2304	36864
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	2560	40960
B	176	177	178	179	180	181	132	183	184	185	186	187	188	189	190	191	2816	45056
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	3072	49152
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	3328	53248
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	3584	57344
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	3840	61440

5		4		3		2		1		0	
HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0	0	0	0	0	0	0	0	0	0
1	1,048,576	1	65,536	1	4,096	1	256	1	16	1	1
2	2,097,152	2	131,072	2	8,192	2	512	2	32	2	2
3	3,145,728	3	196,608	3	12,288	3	768	3	48	3	3
4	4,194,304	4	262,144	4	16,384	4	1,024	4	64	4	4
5	5,242,880	5	327,680	5	20,480	5	1,280	5	80	5	5
6	6,291,456	6	393.216	6	24,576	6	1,536	6	96	6	6
7	7,340,032	7	458,752	7	28,672	7	1,792	7	112	7	7
8	8,388,608	8	524,288	8	32,768	8	2,048	8	128	8	8
9	9,437,184	9	589,824	9	36,864	9	2,304	9	144	9	9
A	10,485,760	A	655,360	A	40,960	A	2,560	A	160	A	10
B	11,534,336	B	720,896	B	45,056	B	2,816	B	176	B	11
C	12,582,912	C	786,432	C	49,152	C	3,072	C	192	C	12
D	13,631,488	D	851,968	D	53,248	D	3,328	D	208	D	13
E	14,680,064	E	917,504	E	57,344	E	3,584	E	224	E	14
F	15.728,640	F	983,040	F	61,440	F	3,840	F	240	F	15

BIJLAGE B

ASCII CONVERSIE TABEL

HEX	MSD	0	1	2	3	4	5	6	7
LSD	BITS	000	001	010	011	100	101	110	111
0	0000	NUL	DLE	SPACE	0	@	P	–	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	¦
D	1101	CR	GS	–	=	M	]	m	}
E	1110	SO	RS	.	>	N	∧	n	~
F	1111	SI	US	/	?	O	←	o	DEL

DE ASCII SYMBOLEN

NUL —Null
SOH —Start of Heading
STX —Start of Text
ETX —End of Text
EOT —End of Transmission
ENQ —Enquiry
ACK —Acknowledge
BEL —Bell
BS —Backspace
HT —Horizontal Tabulation
LF —Line Feed
VT —Vertical Tabulation
FF —Form Feed
CR —Carriage Return
SO —Shift Out
SI —Shift In
DLE —Data Link Escape
DC —Device Control
NAK —Negative Acknowledge
SYN —Synchronous Idle
ETB —End of Transmission Block
CAN —Cancel
EM —End of Medium
SUB —Substitute
ESC —Escape
FS —File Separator
GS —Group Separator
RS —Record Separator
US —Unit Separator
SP —Space (Blank)
DEL —Delete

BIJLAGE C

RELATIEVE SPRONG TABELLEN

RELATIEVE SPRONG (VOORWAARTS) TABEL

MSD \ LSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127

RELATIEVE SPRONG (ACHTERWAARTS) TABEL

MSD \ LSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	128	127	126	125	124	123	122	121	120	119	118	117	116	115	114	113
9	112	111	110	109	108	107	106	105	104	103	102	101	100	99	98	97
A	96	95	94	93	92	91	90	89	88	87	86	85	84	83	82	81
B	80	79	78	77	76	75	74	73	72	71	70	69	68	67	66	65
C	64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49
D	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33
E	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17
F	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

BIJLAGE D

DECIMAAL NAAR BCD CONVERSIE

DECIMAL	BCD	DEC	BCD	DEC	BCD
0	0000	10	00010000	90	10010000
1	0001	11	00010001	91	10010001
2	0010	12	00010010	92	10010010
3	0011	13	00010011	93	10010011
4	0100	14	00010100	94	10010100
5	0101	15	00010101	95	10010101
6	0110	16	00010110	96	10010110
7	0111	17	00010111	97	10010111
8	1000	18	00011000	98	10011000
9	1001	19	00011001	99	10011001

BIJLAGE E

Z80 INSTRUCTIE CODES

OBJ CODE	SOURCE STATEMENT	
8E	ADC	A,(HL)
DD8E05	ADC	A,(IX+d)
FD8E05	ADC	A,(IY+d)
8F	ADC	A,A
88	ADC	A,B
89	ADC	A,C
8A	ADC	A,D
8B	ADC	A,E
8C	ADC	A,H
8D	ADC	A,L
CE20	ADC	A,n
ED4A	ADC	HL,BC
ED5A	ADC	HL,DE
ED6A	ADC	HL,HL
ED7A	ADC	HL,SP
86	ADD	A,(HL)
DD8605	ADD	A,(IX+d)
FD8605	ADD	A,(IY+d)
87	ADD	A,A
80	ADD	A,B
81	ADD	A,C
82	ADD	A,D
83	ADD	A,E
84	ADD	A,H
85	ADD	A,L
C620	ADD	A,n
09	ADD	HL,BC
19	ADD	HL,DE
29	ADD	HL,HL
39	ADD	HL,SP
DD09	ADD	IX,BC
DD19	ADD	IX,DE
DD29	ADD	IX,IX
DD39	ADD	IX,SP
FD09	ADD	IY,BC
FD19	ADD	IY,DE
FD29	ADD	IY,IY
FD39	ADD	IY,SP
A6	AND	(HL)
DDA605	AND	(IX+d)
FDA605	AND	(IY+d)
A7	AND	A
A0	AND	B
A1	AND	C
A2	AND	D
A3	AND	E
A4	AND	H
A5	AND	L

OBJ CODE	SOURCE STATEMENT	
E620	AND	n
CB46	BIT	0,(HL)
DDCB0546	BIT	0,(IX+d)
FDCB0546	BIT	0,(IY+d)
CB47	BIT	0,A
CB40	BIT	0,B
CB41	BIT	0,C
CB42	BIT	0,D
CB43	BIT	0,E
CB44	BIT	0,H
CB45	BIT	0,L
CB4E	BIT	1,(HL)
DDCB054E	BIT	1,(IX+d)
FDCB054E	BIT	1,(IY+d)
CB4F	BIT	1,A
CB48	BIT	1,B
CB49	BIT	1,C
CB4A	BIT	1,D
CB4B	BIT	1,E
CB4C	BIT	1,H
CB4D	BIT	1,L
CB56	BIT	2,(HL)
DDCB0556	BIT	2,(IX+d)
FDCB0556	BIT	2,(IY+d)
CB57	BIT	2,A
CB50	BIT	2,B
CB51	BIT	2,C
CB52	BIT	2,D
CB53	BIT	2,E
CB54	BIT	2,H
CB55	BIT	2,L
CB5E	BIT	3,(HL)
DDCB055E	BIT	3,(IX+d)
FDCB055E	BIT	3,(IY+d)
CB5F	BIT	3,A
CB58	BIT	3,B
CB59	BIT	3,C
CB5A	BIT	3,D
CB5B	BIT	3,E
CB5C	BIT	3,H
CB5D	BIT	3,L
CB66	BIT	4,(HL)
DDCB0566	BIT	4,(IX+d)
FDCB0566	BIT	4,(IY+d)
CB67	BIT	4,A
CB60	BIT	4,B
CB61	BIT	4,C
CB62	BIT	4,D

OBJ CODE	SOURCE STATEMENT	
CB63	BIT	4,E
CB64	BIT	4,H
CB65	BIT	4,L
CB6E	BIT	5,(HL)
DDCB056E	BIT	5,(IX+d)
FDCB056E	BIT	5,(IY+d)
CB6F	BIT	5,A
CB68	BIT	5,B
CB69	BIT	5,C
CB6A	BIT	5,D
CB6B	BIT	5,E
CB6C	BIT	5,H
CB6D	BIT	5,L
CB76	BIT	6,(HL)
DDCB0576	BIT	6,(IX+d)
FDCB0576	BIT	6,(IY+d)
CB77	BIT	6,A
CB70	BIT	6,B
CB71	BIT	6,C
CB72	BIT	6,D
CB73	BIT	6,E
CB74	BIT	6,H
CB75	BIT	6,L
CB7E	BIT	7,(HL)
DDCB057E	BIT	7,(IX+d)
FDCB057E	BIT	7,(IY+d)
CB7F	BIT	7,A
CB78	BIT	7,B
CB79	BIT	7,C
CB7A	BIT	7,D
CB7B	BIT	7,E
CB7C	BIT	7,H
CB7D	BIT	7,L
DC8405	CALL	C,nn
FC8405	CALL	M,nn
D48405	CALL	NC,nn
C48405	CALL	NZ,nn
F48405	CALL	P,nn
EC8405	CALL	PE,nn
E48405	CALL	PO,nn
CC8405	CALL	Z,nn
CD8405	CALL	nn
3F	CCF	
BE	CP	(HL)
DDBE05	CP	(IX+d)
FDBE05	CP	(IY+d)
BF	CP	A
B8	CP	B
B9	CP	C
BA	CP	D
BB	CP	E
BC	CP	H
BD	CP	L
FE20	CP	n
EDA9	CPD	
EDB9	CPDR	

OBJ CODE	SOURCE STATEMENT	
EDB1	CPIR	
EDA1	CPI	
2F	CPL	
27	DAA	
35	DEC	(HL)
DD3505	DEC	(IX+d)
FD3505	DEC	(IY+d)
3D	DEC	A
05	DEC	B
0B	DEC	BC
0D	DEC	C
15	DEC	D
1B	DEC	DE
1D	DEC	E
25	DEC	H
2B	DEC	HL
DD2B	DEC	IX
FD2B	DEC	IY
2D	DEC	L
3B	DEC	SP
F3	DI	
102E	DJNZ	e
FB	EI	
E3	EX	(SP),HL
DDE3	EX	(SP),IX
FDE3	EX	(SP),IY
08	EX	AF,AF'
EB	EX	DE,HL
D9	EXX	
76	HALT	
ED46	IM	0
ED56	IM	1
ED5E	IM	2
ED78	IN	A,(C)
ED40	IN	B,(C)
ED48	IN	C,(C)
ED50	IN	D,(C)
ED58	IN	E,(C)
ED60	IN	H,(C)
ED68	IN	L,(C)
34	INC	(HL)
DD3405	INC	(IX+d)
FD3405	INC	(IY+d)
3C	INC	A
04	INC	B
03	INC	BC
0C	INC	C
14	INC	D
13	INC	DE
1C	INC	E
24	INC	H
23	INC	HL
DD23	INC	IX
FD23	INC	IY
2C	INC	L
33	INC	SP
DB20	IN	A,(n)

OBJ CODE	SOURCE STATEMENT	
EDAA	IND	
EDBA	INDR	
EDA2	INI	
EDB2	INIR	
C38405	JP	nn
E9	JP	(HL)
DDE9	JP	(IX)
FDE9	JP	(IY)
DA8405	JP	C,nn
FA8405	JP	M,nn
D28405	JP	NC,nn
C28405	JP	NZ,nn
F28405	JP	P,nn
EA8405	JP	PE,nn
E28405	JP	PO,nn
CA8405	JP	Z,nn
382E	JR	C,e
302E	JR	NC,e
202E	JR	NZ,e
282E	JR	Z,e
182E	JR	e
02	LD	(BC),A
12	LD	(DE),A
77	LD	(HL),A
70	LD	(HL),B
71	LD	(HL),C
72	LD	(HL),D
73	LD	(HL),E
74	LD	(HL),H
75	LD	(HL),L
3620	LD	(HL),n
DD7705	LD	(IX+d),A
DD7005	LD	(IX+d),B
DD7105	LD	(IX+d),C
DD7205	LD	(IX+d),D
DD7305	LD	(IX+d),E
DD7405	LD	(IX+d),H
DD7505	LD	(IX+d),L
DD360520	LD	(IX+d),n
FD7705	LD	(IY+d),A
FD7005	LD	(IY+d),B
FD7105	LD	(IY+d),C
FD7205	LD	(IY+d),D
FD7305	LD	(IY+d),E
FD7405	LD	(IY+d),H
FD7505	LD	(IY+d),L
FD360520	LD	(IY+d),n
328405	LD	(nn),A
ED438405	LD	(nn),BC
ED538405	LD	(nn),DE
228405	LD	(nn),HL
DD228405	LD	(nn),IX
FD228405	LD	(nn),IY
ED738405	LD	(nn),SP
0A	LD	A,(BC)
1A	LD	A,(DE)
7E	LD	A,(HL)

OBJ CODE	SOURCE STATEMENT	
DD7E05	LD	A,(IX+d)
FD7E05	LD	A,(IY+d)
3A8405	LD	A,(nn)
7F	LD	A,A
78	LD	A,B
79	LD	A,C
7A	LD	A,D
7B	LD	A,E
7C	LD	A,H
ED57	LD	A,I
7D	LD	A,L
3E20	LD	A,n
ED5F	LD	A,R
46	LD	B,(HL)
DD4605	LD	B,(IX+d)
FD4605	LD	B,(IY+d)
47	LD	B,A
40	LD	B,B
41	LD	B,C
42	LD	B,D
43	LD	B,E
44	LD	B,H
45	LD	B,L
0620	LD	B,n
ED4B8405	LD	BC,(nn)
018405	LD	BC,nn
4E	LD	C,(HL)
DD4E05	LD	C,(IX+d)
FD4E05	LD	C,(IY+d)
4F	LD	C,A
48	LD	C,B
49	LD	C,C
4A	LD	C,D
4B	LD	C,E
4C	LD	C,H
4D	LD	C,L
0E20	LD	C,n
56	LD	D,(HL)
DD5605	LD	D,(IX+d)
FD5605	LD	D,(IY+d)
57	LD	D,A
50	LD	D,B
51	LD	D,C
52	LD	D,D
53	LD	D,E
54	LD	D,H
55	LD	D,L
1620	LD	D,n
ED5B8405	LD	DE,(nn)
118405	LD	DE,nn
5E	LD	E,(HL)
DD5E05	LD	E,(IX+d)
FD5E05	LD	E,(IY+d)
5F	LD	E,A
58	LD	E,B
59	LD	E,C
5A	LD	E,D

OBJ CODE	SOURCE STATEMENT	
5B	LD	E,E
5C	LD	E,H
5D	LD	E,L
1E20	LD	E,n
66	LD	H,(HL)
DD6605	LD	H,(IX+d)
FD6605	LD	H,(IY+d)
67	LD	H,A
60	LD	H,B
61	LD	H,C
62	LD	H,D
63	LD	H,E
64	LD	H,H
65	LD	H,L
2620	LD	H,n
2A8405	LD	HL,(nn)
218405	LD	HL,nn
ED47	LD	I,A
DD2A8405	LD	IX,(nn)
DD218405	LD	IX,nn
FD2A8405	LD	IY,(nn)
FD218405	LD	IY,nn
6E	LD	L,(HL)
DD6E05	LD	L,(IX+d)
FD6E05	LD	L,(IY+d)
6F	LD	L,A
68	LD	L,B
69	LD	L,C
6A	LD	L,D
6B	LD	L,E
6C	LD	L,H
6D	LD	L,L
2E20	LD	L,n
ED4F	LD	R,A
ED7B8405	LD	SP,(nn)
F9	LD	SP,HL
DDF9	LD	SP,IX
FDF9	LD	SP,IY
318405	LD	SP,nn
EDA8	LDD	
EDB8	LDDR	
EDA0	LDI	
EDB0	LDIR	
ED44	NEG	
00	NOP	
B6	OR	(HL)
DDB605	OR	(IX+d)
FDB605	OR	(IY+d)
B7	OR	A
B0	OR	B
B1	OR	C
B2	OR	D
B3	OR	E
B4	OR	H
B5	OR	L
F620	OR	n
EDBB	OTDR	

OBJ CODE	SOURCE STATEMENT	
EDB3	OTIR	
ED79	OUT	(C),A
ED41	OUT	(C),B
ED49	OUT	(C),C
ED51	OUT	(C),D
ED59	OUT	(C),E
ED61	OUT	(C),H
ED69	OUT	(C),L
D320	OUT	(n),A
EDAB	OUTD	
EDA3	OUTI	
F1	POP	AF
C1	POP	BC
D1	POP	DE
E1	POP	HL
DDE1	POP	IX
FDE1	POP	IY
F5	PUSH	AF
C5	PUSH	BC
D5	PUSH	DE
E5	PUSH	HL
DDE5	PUSH	IX
FDE5	PUSH	IY
CB86	RES	0,(HL)
DDCB0586	RES	0,(IX+d)
FDCB0586	RES	0,(IY+d)
CB87	RES	0,A
CB80	RES	0,B
CB81	RES	0,C
CB82	RES	0,D
CB83	RES	0,E
CB84	RES	0,H
CB85	RES	0,L
CB8E	RES	1,(HL)
DDCB058E	RES	1,(IX+d)
FDCB058E	RES	1,(IY+d)
CB8F	RES	1,A
CB88	RES	1,B
CB89	RES	1,C
CB8A	RES	1,D
CB8B	RES	1,E
CB8C	RES	1,H
CB8D	RES	1,L
CB96	RES	2,(HL)
DDCB0596	RES	2,(IX+d)
FDCB0596	RES	2,(IY+d)
CB97	RES	2,A
CB90	RES	2,B
CB91	RES	2,C
CB92	RES	2,D
CB93	RES	2,E
CB94	RES	2,H
CB95	RES	2,L
CB9E	RES	3,(HL)
DDCB059E	RES	3,(IX+d)
FDCB059E	RES	3,(IY+d)

OBJ CODE	SOURCE STATEMENT	
CB9F	RES	3,A
CB98	RES	3,B
CB99	RES	3,C
CB9A	RES	3,D
CB9B	RES	3,E
CB9C	RES	3,H
CB9D	RES	3,L
CBA6	RES	4,(HL)
DDCB05A6	RES	4,(IX+d)
FDCB05A6	RES	4,(IY+d)
CBA7	RES	4,A
CBA0	RES	4,B
CBA1	RES	4,C
CBA2	RES	4,D
DBA3	RES	4,E
CBA4	RES	4,H
CBA5	RES	4,L
CBAE	RES	5,(HL)
DDCB05AE	RES	5,(IX+d)
FDCB05AE	RES	5,(IY+d)
CBAF	RES	5,A
CBA8	RES	5,B
CBA9	RES	5,C
CBAA	RES	5,D
CBAB	RES	5,E
CBAC	RES	5,H
CBAD	RES	5,L
CBB6	RES	6,(HL)
DDCB05B6	RES	6,(IX+d)
FDCB05B6	RES	6,(IY+d)
CBB7	RES	6,A
CBB0	RES	6,B
CBB1	RES	6,C
CBB2	RES	6,D
CBB3	RES	6,E
CBB4	RES	6,H
CBB5	RES	6,L
CBBE	RES	7,(HL)
DDCB05BE	RES	7,(IX+d)
FDCB05BE	RES	7,(IY+d)
CBBF	RES	7,A
CBB8	RES	7,B
CBB9	RES	7,C
CBBA	RES	7,D
CBBB	RES	7,E
CBBC	RES	7,H
CBBD	RES	7,L
C9	RET	
D8	RET	C
F8	RET	M
D0	RET	NC
C0	RET	NZ
F0	RET	P
E8	RET	PE
E0	RET	P0
C8	RET	Z

OBJ CODE	SOURCE STATEMENT	
ED4D	RETI	
ED45	RETN	
CB16	RL	(HL)
DDCB0516	RL	(IX+d)
FDCB0516	RL	(IY+d)
CB17	RL	A
CB10	RL	B
CB11	RL	C
CB12	RL	D
CB13	RL	E
CB14	RL	H
CB15	RL	L
17	RLA	
CB06	RLC	(HL)
DDCB0506	RLC	(IX+d)
FDCB0506	RLC	(IY+d)
CB07	RLC	A
CB00	RLC	B
CB01	RLC	C
CB02	RLC	D
CB03	RLC	E
CB04	RLC	H
CB05	RLC	L
07	RLCA	
ED6F	RLD	
CB1E	RR	(HL)
DDCB051E	RR	(IX+d)
FDCB051E	RR	(IY+d)
CB1F	RR	A
CB18	RR	B
CB19	RR	C
CB1A	RR	D
CB1B	RR	E
CB1C	RR	H
CB1D	RR	L
1F	RRA	
CB0E	RRC	(HL)
DDCB050E	RRC	(IX+d)
FDCB050E	RRC	(IY+d)
CB0F	RRC	A
CB08	RRC	B
CB09	RRC	C
CB0A	RRC	D
CB0B	RRC	E
CB0C	RRC	H
CB0D	RRC	L
OF	RRCA	
ED67	RRD	
C7	RST	00H
CF	RST	08H
D7	RST	10H
DF	RST	18H
E7	RST	20H
EF	RST	28H
F7	RST	30H
FF	RST	38H
DE20	SBC	A,n

OBJ CODE	SOURCE STATEMENT	
9E	SBC	A,(HL)
DD9E05	SBC	A,(IX+d)
FD9E05	SBC	A,(IY+d)
9F	SBC	A,A
98	SBC	A,B
99	SBC	A,C
9A	SBC	A,D
9B	SBC	A,E
9C	SBC	A,H
9D	SBC	A,L
ED42	SBC	HL,BC
ED52	SBC	HL,DE
ED62	SBC	HL,HL
ED72	SBC	HL,SP
37	SCF	
CBC6	SET	0,(HL)
DDCB05C6	SET	0,(IX+d)
FDCB05C6	SET	0,(IY+d)
CBC7	SET	0,A
CBC0	SET	0,B
CBC1	SET	0,C
CBC2	SET	0,D
CBC3	SET	0,E
CBC4	SET	0,H
CBC5	SET	0,L
CBCE	SET	1,(HL)
DDCB05CE	SET	1,(IX+d)
FDCB05CE	SET	1,(IY+d)
CBCF	SET	1,A
CBC8	SET	1,B
CBC9	SET	1,C
CBCA	SET	1,D
CBCB	SET	1,E
CBCC	SET	1,H
CBCD	SET	1,L
CBD6	SET	2,(HL)
DDCB05D6	SET	2,(IX+d)
FDCB05D6	SET	2,(IY+d)
CBD7	SET	2,A
CBD0	SET	2,B
CBD1	SET	2,C
CBD2	SET	2,D
CBD3	SET	2,E
CBD4	SET	2,H
CBD5	SET	2,L
CBD8	SET	3,B
CBDE	SET	3,(HL)
DDCB05DE	SET	3,(IX+d)
FDCB05DE	SET	3,(IY+d)
CBDF	SET	3,A
CBD9	SET	3,C
CBDA	SET	3,D
CBDB	SET	3,E
CBDC	SET	3,H
CBDD	SET	3,L
CBE6	SET	4,(HL)

OBJ CODE	SOURCE STATEMENT	
DDCB05E6	SET	4,(IX+d)
FDCB05E6	SET	4,(IY+d)
CBE7	SET	4,A
CBE0	SET	4,B
CBE1	SET	4,C
CBE2	SET	4,D
CBE3	SET	4,E
CBE4	SET	4,H
CBE5	SET	4,L
CBEE	SET	5,(HL)
DDCB05EE	SET	5,(IX+d)
FDCB05EE	SET	5,(IY+d)
CBEF	SET	5,A
CBE8	SET	5,B
CBE9	SET	5,C
CBEA	SET	5,D
CBEB	SET	5,E
CBEC	SET	5,H
CBED	SET	5,L
CBF6	SET	6,(HL)
DDCB05F6	SET	6,(IX+d)
FDCB05F6	SET	6,(IY+d)
CBF7	SET	6,A
CBF0	SET	6,B
CBF1	SET	6,C
CBF2	SET	6,D
CBF3	SET	6,E
CBF4	SET	6,H
CBF5	SET	6,L
CBFE	SET	7,(HL)
DDCB05FE	SET	7,(IX+d)
FDCB05FE	SET	7,(IY+d)
CBFF	SET	7,A
CBF8	SET	7,B
CBF9	SET	7,C
CBFA	SET	7,D
CBFB	SET	7,E
CBFC	SET	7,H
CBFD	SET	7,L
CB26	SLA	(HL)
DDCB0526	SLA	(IX+d)
FDCB0526	SLA	(IY+d)
CB27	SLA	A
CB20	SLA	B
CB21	SLA	C
CB22	SLA	D
CB23	SLA	E
CB24	SLA	H
CB25	SLA	L
CB2E	SRA	(HL)
DDCB052E	SRA	(IX+d)
FDCB052E	SRA	(IY+d)
CB2F	SRA	A
CB28	SRA	B
CB29	SRA	C
CB2A	SRA	D

OBJ CODE	SOURCE STATEMENT	
CB2B	SRA	E
CB2C	SRA	H
CB2D	SRA	L
CB3E	SRL	(HL)
DDCB053E	SRL	(IX+d)
FDCB053E	SRL	(IY+d)
CB3F	SRL	A
CB38	SRL	B
CB39	SRL	C
CB3A	SRL	D
CB3B	SRL	E
CB3C	SRL	H
CB3D	SRL	L
96	SUB	(HL)
DD9605	SUB	(IX+d)
FD9605	SUB	(IY+d)
97	SUB	A
90	SUB	B
91	SUB	C
92	SUB	D
93	SUB	E
94	SUB	H
95	SUB	L
D620	SUB	n
AE	XOR	(HL)
DDAE05	XOR	(IX+d)
FDAE05	XOR	(IY+d)
AF	XOR	A
A8	XOR	B
A9	XOR	C
AA	XOR	D
AB	XOR	E
AC	XOR	H
AD	XOR	L
EE20	XOR	n

(Courtesy of Zilog Inc.)

BIJLAGE F

Z80 naar 8080 EQUIVALENTEN

Z80	8080	Z80	8080	Z80	8080
ADC A, (HL)	ADC M	EX (SP), HL	XTHL	OR n	ORI [B2]
ADC A, n	ACI [B2]	HALT	HLT	OR r	ORA r
ADC A, r	ADC r	IN A, (n)	IN [B2]	OR (HL)	ORA M
ADD A, (HL)	ADD M	INC BC	INX B	OUT (n), A	OUT [B2]
ADD A, n	ADI [B2]	INC DE	INX D	POP AF	POP PSW
ADD A, r	ADD r	INC HL	INX H	POP BC	POP B
ADD HL, BC	DAD B	INC r	INR r	POP DE	POP D
ADD HL, DE	DAD D	INC SP	INX SP	POP HL	POP H
ADD HL, HL	DAD H	INC (HL)	INR M	PUSH AF	PUSH PSW
ADD HL, SP	DAD SP	JP C, nn	JC [B2] [B3]	PUSH BC	PUSH B
AND n	ANI [B2]	JP M, nn	JM [B2][B3]	PUSH DE	PUSH D
AND r	ANA r	JP NC, nn	JNC [B2] [B3]	PUSH HL	PUSH H
AND (HL)	ANA M	JP nn	JMP [B2] [B3]	RET	RET
CALL C, nn	CC [B2] [B3]	JP NZ, nn	JNZ [B2] [B3]	RET C	RC
CALL M, nn	CM [B2] [B3]	JP P, nn	JP [B2] [B3]	RET M	RM
CALL NC, nn	CNC [B2] [B3]	JP PE, nn	JPE [B2][B3]	RET NC	RNC
CALL nn	CALL	JP PO, nn	JPO [B2][B3]	RET NZ	RNZ
CALL NZ, nn	CNZ [B2] [B3]	JP Z, nn	JZ [B2] [B3]	RET P	RP
CALL P, nn	CP [B2] [B3]	JP (HL)	PCHL	RET PE	RPE
CALL PE, nn	CPE [B2] [B3]	LD A, (DE)	LDAX	RET PO	RPO
CALL PO, nn	CPO [B2] [B3]	LDA, (nn)	LDA [B2] [B3]	RET Z	RZ
CALL Z, nn	CZ [B2] [B3]	LD DE, nn	LXID, [B2] [B3]	RLA	RAL
CCF	CMC	LD SP, nn	LXI SP, [B2] [B3]	RLCA	RLC
CP r	CMP r	LD (BC), A	STAX B	RRA	RAR
CP (HL)	CMP M	LD (DE), A	STAX D	RRCA	RRC
CPL	CMA	LD (HL), r	MOV M, r	RST P	RST P
CP n	CPI [B2]	LD (nn), A	STA [B2] [B3]	SBC A, (HL)	SBB M
DAA	DAA	LD (nn), HL	SHLD [B2] [B3]	SBC A, n	SBI [B2]
DEC BC	DCX B	LD A, (BC)	LDAX B	SBC A, r	SBB r
DEC DE	DCX D	LD BC, nn	LXIB, [B2] [B3]	SCF	STC
DEC HL	DCX H	LD HL, (nn)	LHLD [B2] [B3]	SUB n	SUI [B2]
DEC r	DCR r	LD HL, nn	LXI H [B2] [B3]	SUB r	SUB r
DEC SP	DCX SP	LD r, (HC)	MOV 1, M	SUB (HL)	SUB M
DEC (HL)	DCR M	LD r, n	MVI r, [B2]	XOR n	XRI [B2]
DI	DI	LD r, r'	MOV r1, r2	XOR r	XRA r
EI	EI	LD SP, HL	SPHL	XOR (HL)	XRA M
EX DE, HL	XCHG	NOP	NOP		

BIJLAGE G

8080 naar Z80 EQUIVALENTEN

8080	Z80	8080	Z80	8080	Z80
ACI [B2]	ADC A, n	IN [B2]	IN A, (n)	POP H	POP HL
ADC M	ADC A, (HL)	INR M	INC (HL)	POP PSW	POP AF
ADC r	ADC A, r	INR r	INC r	PUSH B	PUSH BC
ADD M	ADD A, (HL)	INX B	INC BC	PUSH D	PUSH DE
ADD r	ADD A, r	INX D	INC DE	PUSH H	PUSH HL
ADI [B2]	ADD A, n	INX H	INC HL	PUSH PSW	PUSH AF
ANA M	AND (HL)	INX SP	INC SP	RAL	RLA
ANA r	AND r	JC [B2] [B3]	JP C, nn	RAR	RRA
ANI [B2]	AND n	JM [B2] [B3]	JP M, nn	RC	RET C
CALL	CALL nn	JMP [B2] [B3]	JP nn	RET	RET
CC [B2] [B3]	CALL C, nn	JNC [B2] [B3]	JP NC, nn	RLC	RLCA
CM [B2] [B3]	CALL M, nn	JNZ [B2] [B3]	JP NZ, nn	RM	RET M
CMA	CPL	JP [B2] [B3]	JP P, nn	RNC	RET NC
CMC	CCF	JPE [B2] [B3]	JP PE, nn	RNZ	RET NZ
CMP M	CP (HL)	JPO [B2] [B3]	JP PO, nn	RP	RET P
CMP r	CP r	JZ [B2] [B3]	JP Z, nn	RPE	RET PE
CNC [B2] [B3]	CALL NC, nn	LDA [B2] [B3]	LD A, (nn)	RPO	RET PO
CNZ [B2] [B3]	CALL NZ, nn	LDAX B	LD A, (BC)	RRC	RRCA
CP [B2] [B3]	CALL P, nn	LDAX D	LD A, (DE)	RST	RST P
CPE [B2] [B3]	CALL PE, nn	LH LD [B2] [B3]	LD HL, (nn)	RZ	RET Z
CPI [B2]	CP n	LXI B [B2] [B3]	LD BC, nn	SBB M	SBC A, (HL)
CPO [B2] [B3]	CALL PO, nn	LDID [B2] [B3]	LD DE, nn	SBB r	SBC A, r
CZ [B2] [B3]	CALL Z, nn	LXI H [B2] [B3]	LD HL, nn	SBI [B2]	SBC A, n
DAA	DAA	LXI SP [B2] [B3]	LD SP, nn	SHLD [B2] [B3]	LD (nn), HL
DAD B	ADD HL, BC	MOV M, r	LD (HL), r	SPHL	LD SP, HL
DAD D	ADD HL, DE	MOV r, M	LD r, (HL)	STA [B2] [B3]	LD (nn), A
DAD H	ADD HL, HL	MOV r1, r2	LD r, r'	STAX B	LD (BC), A
DAD SP	ADD HL, SP	MVI M	LD (HL), n	STAX D	LD (DE), A
DCR M	DEC (HL)	MVI r [B2]	LD r, n	STC	SCF
DCR r	DEC r	NOP	NOP	SUB M	SUB (HL)
DCX B	DEC BC	ORA M	OR (HL)	SUB r	SUB r
DCX D	DEC DE	ORA r	OR r	SUI [B2]	SUB n
DCX H	DEC HL	ORI [B2]	OR n	XCHG	EX DE, HL
DCX SP	DEC SP	OUT [B2]	OUT (n), A	XRA M	XOR (HL)
DI	DI	PCHL	JP (HL)	XRA r	XOR r
EI	EI	POP B	POP BC	XRI [B2]	XOR n
HALT	HLT	POP D	POP DE	XTHL	EX (SP), HL

INDEX

The SYBEX Library

in English:

BASIC PROGRAMS FOR SCIENTISTS AND ENGINEERS
by Alan R. Miller 340 pp., 120 illustr., Ref. B240
This second book in the "Programs for Scientists and Engineers" series provides a library of problem solving programs while developing proficiency in BASIC.

INSIDE BASIC GAMES

By Richard Mateosian 350 pp., 240 illustr., Ref. B245
Teaches interactive BASIC programming through games. Games are written in Microsoft BASIC and can run on the TRS-80, APPLE II and PET/CBM.

FIFTY BASIC EXERCISES

by J. P. Lamoitier 240 pp., 195 illustr., Ref. B250

YOUR FIRST COMPUTER

by Rodnay Zaks 260 pp., 150 illustr., Ref. C200A
The most popular introduction to small computers and their peripherals: what they do and how to buy one.

DON'T (or How to Care for Your Computer)

by Rodnay Zaks 220 pp., 100 illustr., Ref. C400
The correct way to handle and care for all elements of a computer system including what to do when something doesn't work.

INTRODUCTION TO WORD PROCESSING

by Haly Glatzer 200 pp., 70 illustr., Ref. W101
Explains in plain language what a word processor can do, how it improves productivity, how to use a word processor and how to buy one wisely.

INTRODUCTION TO WORDSTAR

by Arthur Naiman 200 pp., 30 illustr., Ref. W110
Makes it easy to learn how to use WordStar, a powerful word processing program for personal computers.

FROM CHIPS TO SYSTEMS: AN INTRODUCTION TO MICROPROCESSORS

by Rodnay Zaks 560 pp., 255 illustr., Ref. C201A
A simple and comprehensive introduction to microprocessors from both a hardware and software standpoint: what they are, how they operate, how to assemble them into a complete system.

MICROPROCESSOR INTERFACING TECHNIQUES

by Rodnay Zaks and Austin Lesea 460 pp., 400 illustr., Ref. C207
Complete hardware and software interconnect techniques including D to A conversion, peripherals, standard buses and troubleshooting.

PROGRAMMING THE 6502

by Rodnay Zaks 390 pp., 160 illustr., Ref. C202
Assembly language programming for the 6502, from basic concepts to advanced data structures.

6502 APPLICATIONS BOOK

by Rodnay Zaks 280 pp., 205 illustr., Ref. D302
Real life application techniques: the input/output book for the 6502.

6502 GAMES

by Rodnay Zaks 300 pp., 140 illustr., Ref. G402
Third in the 6502 series. Teaches more advanced programming techniques, using games as a framework for learning.

PROGRAMMING THE Z80

by Rodnay Zaks 620 pp., 200 illustr., Ref. C280
A complete course in programming the Z80 microprocessor and a thorough introduction to assembly language.

PROGRAMMING THE Z8000

by Richard Mateosian 300 pp., 125 illustr., Ref. C281
How to program the Z8000 16-bit microprocessor. Includes a description of the architecture and function of the Z8000 and its family of support chips.

THE CP/M HANDBOOK (with MP/M)

by Rodnay Zaks 330 pp., 100 illustr., Ref. C300
An indispensable reference and guide to CP/M – the most widely used operating system for small computers.

INTRODUCTION TO PASCAL (Including UCSD PASCAL)

by Rodnay Zaks 420 pp., 130 illustr., Ref. P310
A step-by-step introduction for anyone wanting to learn the Pascal language. Describes UCSD and Standard Pascals. No technical background is assumed.

THE PASCAL HANDBOOK

by Jacques Tiberghien 490 pp., 350 illustr., Ref. P320
A dictionary of the Pascal language, defining every reserved word, operator, procedure and function found in all major versions of Pascal.

PASCAL PROGRAMS FOR SCIENTISTS AND ENGINEERS

by Alan Miller 400 pp., 80 illustr., Ref. P340
A comprehensive collection of frequently used algorithms for scientific and technical applications, programmed in Pascal. Includes such programs as curve-fitting, integrals and statistical techniques.

50 PASCAL PROGRAMS

by Rudolph Langer and Rodnay Zaks 275 pp., 90 illustr., Ref. P350
A collection of 50 Pascal programs ranging from mathematics to business and games programs. Explains programming techniques and provides actual practice.

APPLE PASCAL GAMES

by Douglas Hergert and Joseph T. Kalash 380 pp., 40 illustr., Ref. P360
A collection of the most popular computer games in Pascal challenging the reader not only to play but to investigate how games are implemented on the computer.

INTRODUCTION TO UCSD PASCAL SYSTEMS

by Charles T. Grant and Jon Butah 300 pp., 110 illustr., Ref. P370
A simple, clear introduction to the UCSD Pascal Operating System for beginners through experienced programmers.

INTERNATIONAL MICROCOMPUTER DICTIONARY

140 pp., Ref. X2
All the definitions and acronyms of microcomputer jargon defined in a handy pocket-size edition. Includes translations of the most popular terms into ten languages.

MICROPROGRAMMED APL IMPLEMENTATION

by Rodnay Zaks 350 pp., Ref. Z10
An expert-level text presenting the complete conceptual analysis and design of an APL interpreter, and actual listings of the microcode.

SELF STUDY COURSES

Recorded live at seminars given by recognized professionals in the microprocessor field.

INTRODUCTORY SHORT COURSES:

Each includes two cassettes plus special coordinated workbook ($2^1/_2$ hours).

S10–INTRODUCTION TO PERSONAL AND BUSINESS–COMPUTING

A comprehensive introduction to small computer systems for those planning to use or buy one, including peripherals and pitfalls.

S1–INTRODUCTION TO MICROPROCESSORS

How microprocessors work, including basic concepts, applications, advantages and disadvantages.

S2–PROGRAMMING MICROPROCESSORS

The companion to S1. How to program any standard microprocessor, and how it operates internally. Requires a basic understanding of microprocessors.

S3-DESIGNING A MICROPROCESSOR SYSTEM

Learn how to interconnect a complete system, wire by wire. Techniques discussed are applicable to all standard microprocessors.

INTRODUCTORY COMPREHENSIVE COURSES:

Each includes a 300-500 page seminar book and seven or eight C90 cassettes.

SB1–MICROPROCESSORS

This seminar teaches all aspects of microprocessors: from the operation of an MPU to the complete interconnect of a system. The basic hardware course (12 hours).

SB2–MICROPROCESSOR PROGRAMMING

The basic software course: step by step through all the important aspects of microcomputer programming (10 hours).

ADVANCED COURSES:

Each includes a 300-500 page workbook and three or four C90 cassettes.

SB3–SEVERE ENVIRONMENT/MILITARY MICROPROCESSOR SYSTEMS

Complete discussion of constraints, techniques and systems for severe environment applications, including Hughes, Raytheon, Actron and other militarized systems (6 hours).

SB5–BIT-SLICE

Learn how to build a complete system with bit slices. Also examines innovative applications of bit slice techniques (6 hours).

SB6–INDUSTRIAL MICROPROCESSOR SYSTEMS

Seminar examines actual industrial hardware and software techniques, components, programs and cost ($4^1/_2$ hours).

SB7–MICROPROCESSOR INTERFACING

Explains how to assemble, interface and interconnect a system (6 hours).

SOFTWARE

BAS 65™ CROSS-ASSEMBLER IN BASIC

8" diskette, Ref. BAS 65
A complete assembler for the 6502, written in standard Microsoft BASIC under CP/M®.

8080 SIMULATORS

Turns any 6502 into an 8080. Two versions are available for APPLE II.
APPLE II cassette, Ref. S6580-APL(T)
APPLE II diskette, Ref. S6580-APL(D)

VRAAG EEN COMPLETE CATALOGUS AAN BIJ:

SYBEX INC.
2344 Sixth Street
Berkeley,
California 94710
Tel: (415) 848-8233
Telex: 336311

SYBEX-EUROPE
4 Place Félix Eboué
75583 Paris Cedex 12
Tel: 1/347-30-20
Telex: 211801

SYBEX-VERLAG
Heyestr. 22
4000 Düsseldorf 12
West Germany
Tel: (0211) 287066
Telex: 8 588 163

Imprimé en Hollande
Drukkerij Salland b.v.
Postbus 23
7400 GA Deventer.

Dépôt légal:
février 1982

Programmeren van de Z80

is een boek dat niet alleen geschikt is om het programmeren te leren, maar kan ook als naslagwerk gebruikt worden. Het bevat niet alleen de basisbegrippen m.b.t. het programmeren, maar ook gevorderde bewerkingen op data structuren.
Bovendien is er een beschrijving in opgenomen van alle Z80 instructies. De interne werking van de Z80 komt eveneens aan de orde. Ook voor de lezer, die de grondbeginselen van het programmeren al onder de knie heeft, maar die de Z80 nog niet kent, is dit boek een waardevolle aanwinst.
Dank zij de ervaring die de auteur in het onderwijs en met het programmeren heeft opgedaan, is dit boek tot stand gekomen. Helder en in eenvoudige termen worden alle principes, beginnend bij de eenvoudige, uit de doeken gedaan. Langzaam maar zeker bouwt de lezer zijn kennis op, niet alleen over de Z80, maar ook over de manier waarop een microprocessor, zoals de Z80, de instructies verwerkt. De informatie stromen over de diverse bussen en tussen de registers van de Z80 zijn voor de lezer duidelijk te volgen. Voor het goed en efficient programmeren van een microprocessor is deze kennis onontbeerlijk.
Omdat programmeren niet alleen bestaat uit het coderen van een algoritme in een programmeertaal, maar ook uit het toepassen van de juiste data structuren, is aan dit onderwerp een uitgebreid hoofdstuk gewijd. Besproken worden lijsten, tabellen, binaire bomen en de bijbehorende algoritmen.
Na het lezen van dit boek moet de lezer in staat zijn voor de meeste practische toepassingen een programma te schijven.

Behandeld worden onder meer:
- Z80 Hardware Organisatie
- Input/Output Technieken
- de Complete Instructie Set
- Z80 adresserings modes
- Data Structuren - Theorie en Ontwerp
- Voorbeelden van Toepassingen

Over de schrijver

Dr. Rodnay Zaks is sinds de microprocessor gebruikt wordt in de industrie daarbij betrokken geweest.
Hij is de schrijver van een aantal best-sellers over microprocessors, en heeft microprocessor cursussen gegeven aan meer dan 5000 personen. Aan de Universiteit van California, Berkeley heeft hij een graad gehaald in computer wetenschappen, en hij is lid van ACM en IEEE.